UNIVERSITY OF CAPE TOWN

MINOR DISSERTATION PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MSC. DATA SCIENCE

AT THE FACULTY OF SCIENCE, DEPARTMENT OF STATISTICAL SCIENCE AND COMPUTER SCIENCE

Automated Feature Synthesis on Big Data using Cloud Computing Resources

Author
Vanessa SAKER

Supervisor
Associate Professor.
Sonia BERMAN

October 8, 2020

PLAGIARISM DECLARATION

I know the meaning of plagiarism and declare that all of the work in the dissertation (or thesis), save for that which is properly acknowledged, is my own.

Signed by candidate

Abstract

The data analytics process has many time-consuming steps. Combining data that sits in a relational database warehouse into a single relation while aggregating important information in a meaningful way and preserving relationships across relations, is complex and time-consuming. This step is exceptionally important as many machine learning algorithms require a single file format as an input (e.g. supervised and unsupervised learning, feature representation and feature learning, etc.). An analyst is required to manually combine relations while generating new, more impactful information points from data during the feature synthesis phase of the feature engineering process that precedes machine learning. Furthermore, the entire process is complicated by Big Data factors such as processing power and distributed data storage.

There is an open-source package, Featuretools, that uses an innovative algorithm called Deep Feature Synthesis to accelerate the feature engineering step. However, when working with Big Data, there are two major limitations. The first is the curse of modularity – Featuretools stores data in-memory to process it and thus, if data is large, it requires a processing unit with a large memory. Secondly, the package is dependent on data stored in a Pandas DataFrame. This makes the use of Featuretools with Big Data tools such as Apache Spark, a challenge.

This dissertation aims to examine the viability and effectiveness of using Featuretools for feature synthesis with Big Data on the cloud computing platform, AWS. Exploring the impact of generated features is a critical first step in solving any data analytics problem. If this can be automated in a distributed Big Data environment with a reasonable investment of time and funds, data analytics exercises will benefit considerably.

In this dissertation, a framework for automated feature synthesis with Big Data is proposed and an experiment conducted to examine its viability. Using this framework, an infrastructure was built to support the process of feature synthesis on AWS that made use of S3 storage buckets, Elastic Cloud Computing services, and an Elastic MapReduce cluster. A dataset of 95 million customers, 34 thousand fraud cases and 5.5 million transactions across three different relations was then loaded into the distributed relational database on the platform. The infrastructure was used to show how the dataset could be prepared to represent a business problem, and Featuretools used to generate a single feature matrix suitable for inclusion in a machine learning pipeline.

The results show that the approach was viable. The feature matrix produced 75 features from 12 input variables and was time efficient with a total end-to-end run time of 3.5 hours and a cost of approximately R 814 (approximately $52). The framework can be applied to a different set of data and allows the analysts to experiment on a small section of the data until a final feature set is decided. They are able to easily scale the feature matrix to the full dataset. This ability to automate feature synthesis, iterate and scale up, will save time in the analytics process while providing a richer feature set for better machine learning results.

Contents

List of Figures

List of Tables

1 Introduction

Data Analytics is the understanding, interrogating and discovery of patterns within data that can inform, describe or support decisions. It is a wide and varied field; from descriptive analytics which can help understand the past to predictive analytics which gives insight into the future and prescriptive analytics which can help optimise decisions, there is immense value in applying analytics to problems. At the root of analytics, lies data. And yet, one of the most underestimated steps in any analytics process is the amount of effort required to process and structure data in preparation for the analytics process. This work proposes a framework for automating the feature synthesis component of data analytics, and describes an investigation into the viability and effectiveness of this framework in the context of Big Data on a distributed cloud computing platform.

1.1 Aims and Motivations

In many cases, data needs to be in a particular structure for analytics to take place. In the case of descriptive analytics, such as management information dashboards and reporting, data is stored in "cubes" (a multi-dimensional array of data) which allows drill down, dicing and roll-up views of a set of data. In the case of supervised machine learning, data is required in a flat-file format - normally a single table, dataframe or array which contains the target variable and a set of variables or features from which the target can be predicted. Unsupervised machine learning algorithms similarly require a flat-file format but do not need a target variable. Both types of machine learning algorithms inherently cannot handle data stored across multiple tables with complex relationships and varying granularity in their structure.

Unfortunately, in industry, especially in large corporations such as banks, insurers, etc. the majority of data is stored in this manner - through complex and governed relational databases (RDBs). Thus, if an industry professional wishes to prepare data for machine learning, they must first compress and aggregate data into a single table. This input into a machine learning algorithm must preserve relationships and granularity in a way that allows the algorithm to understand the important aspects of, and the inherent relationship within, the data.

For instance, in a bank's credit default model, data may be stored on a customer, product and transactional level. This data needs to be aggregated up to a single, pre-determined level which is representative of the problem. If the aim is to predict if a customer would default, the data needs to engineered to a customer level. This would mean rolling up multiple transactions across possible multiple products in a meaningful way that will add value to the prediction algorithm. However, if the aim is predicting the probability of default on a product, the customer data needs to be dispersed across the products while the transactional data is rolled to a product, rather than a customer level.

One of the biggest issues with data science projects in industry is the time it takes to develop and test models. The process of feature engineering can be a time-consuming part of this and often requires domain-specific knowledge. Feature engineering is the process of creating new and meaningful data points from an existing set of data in order to isolate important information and improve a model's performance. Feature synthesis is the process of creating these data points while feature selection selects only the important and meaningful variables. Generally, if the analyst is not familiar with the domain, feature synthesis would require them to interview domain experts to ascertain what potential data points or relationships may be of value and then manually construct features. This is done through coding features individually either within the RDB (relational database) using SQL or within the environment where the machine algorithm will be processed.

For instance, in the above example, the credit assessors or verification employee may have a good gut or instinctual feel about the types of information that would lead to a customer defaulting. Perhaps they have noticed that any application with a business address or no contact number is usually a good indicator of a customer's ability to pay back a loan. The analyst would then have to find a way to represent this in the data by including a flag for incomplete customer contact details. This process can take a great deal of time. The quicker and easier the process of feature engineering can become, the quicker data analytics problems can be solved, while at the same time producing superior solutions.

The centralised benefit to feature engineering focuses on the fact that to achieve a good model it must be based on quality data that is a good representation of the underlying problem to be solved. Feature engineering ensures that the best possible representation of the problem is achieved by squeezing the data to make it fit for purpose through transformations and aggregation of the data. This process often allows algorithms to extract patterns in the data it might have missed. If a model's main aim is prediction, the act of feature engineering can improve the accuracy of a model. Alternatively, if interpolation is the aim, features can help with interpreting what the model considered to be important.

Despite the beneficial impact that feature engineering has and the cost of the time it adds to the overall analytics process, there is not much research in academia or many applied methodologies in the industry that solve this issue. The most well-known tool is Featuretools, an open-source Python package which was developed off the back of an academic master dissertation from MIT in 2015. Now the flagship product of FeatureLabs a DARPA (the US Defence Advanced Research Projects Agency) funded company, it offers an ingenious way to connect and aggregate data over multiple tables in a RDB to a single flat file structure using an algorithm known as Deep Feature Synthesis (DFS).

Besides the amount of time that feature engineering takes, a secondary issue is the processing power that is needed to process data as an input step into a machine learning algorithm - especially when that data is on the Big Data spectrum. Cleaning, imputation of missing values and feature engineering are all crucial steps to produce a high-quality model. To add complexity to this issue, much of the data that companies own is not only stored in a RDB but also

takes advantage of Big Data storage solutions or cloud computing solutions i.e. the RDB where the data is stored, sits over multiple servers in a distributed way. These solutions solve the processing power issue but complicate the use of Featuretools.

Featuretools requires data to be held in memory and relies on the data being stored in a Pandas DataFrame. If a corporate business wishes to use the package on its big, distributed dataset, these limitations become a factor. Firstly, because by definition the distributed data sits across servers or machines and will be too large to pull into memory. And secondly, because on most Big Data platforms, data is stored in objects that are not a Pandas DataFrame.

Because of these challenges, the Featuretools website gives some practical examples illustrating several different problems and how they can be solved. In particular, Koehrsen (2018) illustrates how to predict customer churn using the automated feature engineering package on a large dataset using some cloud platform.

1.2 Research Statement

This dissertation aims to explore the viability and effectiveness of automated feature synthesis for Big Data by designing a framework for producing a matrix of synthesised features that can be input into a machine learning pipeline. If viability and effectiveness does exist, it will allow ease and automation of feature generation for future projects that want to take advantage of Big Data. The emphasis of this dissertation is on setting up a reusable infrastructure and framework and not in generating valuable features for any particular problem.

The data used for this experiment revolves around card fraud at a customer level and consists of: 95 million customer records, 32 000 fraud cases and 5.5 million card purchase transactions which are stored across three tables in a relational database.

In short, this dissertation takes a real-life dataset which has been anonymised and white-labelled, and finds a way to automatically derive features from the dataset which will be suitable for a supervised machine learning problem in an environment that caters for Big Data.

1.3 Objectives and Scope

The objectives of this dissertation are three-fold. The first objective is to research valid options for automated feature synthesis. The second is to set up infrastructure on a cloud-computing resource that will cater to the storage and computation of Big Data. The last objective is to demonstrate a framework that can be used to combine the feature automated synthesis and the infrastructure through an example. In doing so, the effectiveness and viability of the framework can be measured. In order to measure viability and effectiveness, the resulting feature matrix will be examined, the time the framework took to run

measured and the cost of the experiment recorded.

1.4 Limitations

Several related points and items are outside of the scope of this dissertation:

End to end machine learning Data preprocessing and feature synthesis are not a widely explored field of Data Science. On the other hand, feature selection and machine learning algorithms, validation techniques etc. are very well represented. This dissertation thus explores the under represented portion of the problem rather than the actual machine learning. In other words, this work will end where most examples begin - with a set of data suitable for a prediction. The emphasis is on a viable mechanism for automations and the final feature matrix presented would be an example rather than the ultimate list of most important features for the problem. The next steps of data exploration, feature selection etc. are out of scope.

Overview of cloud computing options and functionality Alternative cloud computing platforms were considered but at the same time, a full detailed comparison was not conducted and is not the subject matter of this dissertation. In addition, this dissertation does not serve to give a view of the full functionality of the chosen platform.

1.5 Thesis Outline

Chapter 2 introduces Big Data, overcoming the limitations of processing power and feature engineering. Chapter 3 gives an overview of the infrastructure and framework proposed, while Chapter 4 looks at how each step was implemented in an example involving fraud prediction for a given real-life dataset. Chapter 5 covers the results from the experiment and Chapter 6 reviews conclusions and makes suggestions for further work.

2 Background

This chapter introduces Big Data and cloud computing resources, and then discusses the data analysis process highlighting the importance of feature engineering. It details how feature engineering is handled when dealing with Big Data and available tooling for feature synthesis.

2.1 Big Data

The term "Big Data" is widely cited as being coined in 2005 by Roger Mouglas Press (2013). It is considered to be data that consists of the 3 V's: volume, velocity, and variety (Beyer & Laney, 2012). In other words, Big Data is not only defined in terms of size (either as a large number of instances or features) but can also be defined as a dataset that has a variety of data types and sources (variety) as well as data that is generated at high speed (velocity). In more recent years, an additional two V's have been added: veracity and value. Value is generally derived from Big Data itself (i.e. the outcome of processing Big Data). Veracity is problematic because of the inherent unreliability of Big Data or the sources from which it comes.

When working with Big Data, several issues arise. The 2017 article entitled "Machine Learning with Big Data: Challenges and Approaches" summarises both the broad problems in terms of the definition of Big Data (i.e. in relation to the Big "V"s) and attempts to classify and rate solutions to these issues (L'heureux, Grolinger, Elyamany, & Capretz, 2017). The article raises valid issues and should be considered by any data scientist when dealing with Big Data, however in achieving the aims of the dissertation, only two of these will need to be considered in greater detail: processing performance and feature engineering. The article associates each of these issues to the "volume" aspect of Big Data although, an argument could be made that feature engineering could be a product of disparate sources (variety). For instance, data stored from the call center operational system and data stored from the customer product holding need to be joined and processed to gather a full view of the customer's interactions with the organisation. Possible features that could be added would be how many complaints a customer made per product holding or how the frequency of complaints coincided with the customer acquiring a new product. Feature engineering is the process of synthesizing information through aggregation or transformation of data that could possibly hold important information about the problem under examination. If these created data points do not add value, they may be disregarded or minimised using feature selection or reduction methods.

There is a common misconception that "Big Data" refers only to unstructured data. This is not true. In many cases, all types of Big Data (voluminous, high velocity and high variety) can be stored in relational databases or in a structure. Many companies use MPP (massively parallel processing) database solutions to store and access their data. This dissertation concentrates on data that is both big and structured in nature as this accounts for the majority of corporate

data and is the most important to address in practice.

2.2 Cloud Computing

In datasets of large volume, computing power can become an issue. In Big Datasets where the volume definition exits, the computing power required to do even simple transformation of the data becomes a non-trivial issue. This challenge bears an impact when data pre-processing such as feature engineering is needed on a dataset. Even simple transformations in the preprocessing step such as data normalisation becomes computationally expensive and requires computer resources that are normally outside of the capability of a standalone terminal available to an individual.

Luckily, this is an issue that has not gone unnoticed and solutions do exist to solve it. Techniques in data manipulation (e.g. dimensionality reductions or instance selection), processing manipulations (e.g vertical scaling of computational resources), algorithm modifications and learning paradigms have all had good success in overcoming issues with processing power. In fact, this can be solved with cloud computing services.

Simply put, cloud computing is the delivery of computing services which can include software, storage, servers, networking, and analytical services. These services are generally split into three categories - infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). Cloud computing services are sold on demand (users are charged on a pay as you use basis), they are elastic (i.e. users can have as much or as little of a service as they need and this can be changed as their needs change), and the service is fully managed by the provider (the user is not responsible for the upkeep or maintenance of any of the hard or soft assets). In other words, all that is needed to access the power of cloud computing is a computer, a reliable internet connection, and a credit card. In addition, because cloud services can be either private or public, it appeals to businesses that want to to host in-house systems (private) as well as those that need to offer service to customers (public) and businesses that need to do both (hybrid).

- **Software-as-a-Service** or, as it is sometimes known, on-demand software is a software distribution model that allows users to access, license and use software over the Internet. This software is hosted by the cloud provider or a third party using the cloud to distribute software. For instance SAP, a global software company which specialises in software to help enterprises manage operations and customer relations, uses cloud computing to deliver their flagship customer relationship management (CRM) software, SalesForce, to clients (SAP, 2020).
- **Platform-as-a-Service** is a cloud service that hosts tools for software and application development and provides user access to these through APIs, web portals or gateway software. This allows developers to build and run applications without investing in or owning the infrastructure. In many cases, once the application is built, the cloud provider will also

host the application or software. As an example, Khan Academy, a non-profit educational organisation used the Google App Engine to develop its learning app which is now available on the Google Play and Apple store (GoogleCloud, 2020b).

- **Infrastructure-as-a-Service** provides users with access to information technology infrastructures such as servers, storage, and operating systems. Users can rent what they need without investing or maintaining physical hardware. In addition, IaaS normally provides tools for monitoring, access, security, load balancing, and clustering within the infrastructure and will ensure storage resiliency i.e. backup, replication, and recovery. Some of the companies that offer IaaS are Amazon Web Services (AWS, 2020), Microsoft Azure (Azure, 2020) and Google Compute Engine (GoogleCloud, 2020a).

In essence, cloud computing is perfectly set up to deal with Big Data and the issues that surround its storage, processing and analysis. For companies that are not corporate enterprises with vast resources of money, technology, and skills at their disposal, it is the cheapest and easiest way to gain access to the infrastructure, platforms, and software needed. In fact, even for a large corporate, this may be the best route. However because it is the cheapest or easiest route, does not mean it is either cheap or easy. The aim of this dissertation is to achieve a viable and effective method of automated feature synthesis using cloud computing resources.

Processing power can be a significant issue in Big Datasets and although multiple solutions exist (e.g. data reductions techniques) the most successful variations are in the form of vertical or horizontal scaling of resources. And cloud computing is perfectly positioned to help with either vertical or horizontal scaling as the IaaS service offered by most cloud computing hosts, has access to multiple scalable servers which can be customised to client needs. However in adding horizontal scaling options, the curse of modularity may be triggered.

The "the curse of modularity" as coined by Parker (2012) refers to the fact that in an algorithm or function, it is assumed that the data being processed is held entirely in memory. Translating this algorithm over to a distributed dataset to gain processing performance power creates complexity and may lead to invalidation of algorithms. For instance, many algorithms are based on the assumption that a global or local maximum/minimum exists but when data sits over a distributed system this assumption fails and the optimisation techniques cannot find the optimal points. Parker goes on to point out that parallelisation of algorithms can be a non trivial task. In fact, the deep feature synthesis (DFS) algorithm is subject to the curse of modularity and it is one of the limitations for which this dissertation has to solve.

2.3 Hadoop

In 2004, the world was introduced to MapReduce - a programming model for distributed computing (Dean & Ghemawat, 2004). The two components of the

model (map and reduce) allow for programs written using that approach, to be posted to separate resources, computed at each resource and outputs to be merged from across multiple resources - thus giving programmers a scalable and efficient method of processing data across different computers or servers. The benefit of this was seen by a developer working at the Apache Software Foundation. In 2006, after working with Yahoo and Apache, Doug Cutting started development on a new project called Hadoop which leveraged off the work done on previous Apache projects, MapReduce and a distributed file system developed by Google (GFS - Google File System). In 2011, Apache Hadoop Version 1.0 was launched (White, 2012).

In essence, Hadoop, allows parallel execution of commands and storage of data across a cluster of computing resources.

In the latest release of Hadoop (v3.0), there are three core modules and an additional three modules or sub-projects that support or enhance Hadoop (Foundation, 2019).

- The Hadoop Distributed File System (HDFS) is a file system that stores data across machines or resources. This is a core module.
- MapReduce - the core programming model.
- YARN - Yet Another Resource Negotiator- is a core module that manages the resources and schedules jobs across the cluster.
- Hadoop Common is a module containing the common utilities that support the other Hadoop modules.
- Ozone which is a Hadoop object store.
- Submarine which is a machine learning engine.

The HDFS solves most of the data storage issues that are associated with Big Data by utilising a master/slave architecture across multiple computing resources called a "cluster". A cluster consists of a single "NameNode" and multiple "DataNodes". When data is stored within the HDFS, it is broken up into manageable blocks of data and these blocks are stored on the DataNodes. The mapping of these blocks of data to individual DataNodes is managed by the NameNode which then also takes responsibility for operations such as renaming files and directories etc. In addition, the "NameNode" is responsible for managing the file system namespace and access to the files and data stored on the DataNodes while the DataNodes manage their own storage and read-/write requests. On request from the NameNode, the DataNodes can then create, delete and replicate their blocks of data. This architecture is represented in Figure 1 (Borthakur et al., 2008).

While the HDFS takes care of storage, MapReduce deals with processing across the cluster. In fact, this consists of two separate functions: map and reduce. In a MapReduce "job", the input is split into independent tasks and mapped to the DataNodes which process the request. The results are then fed into the

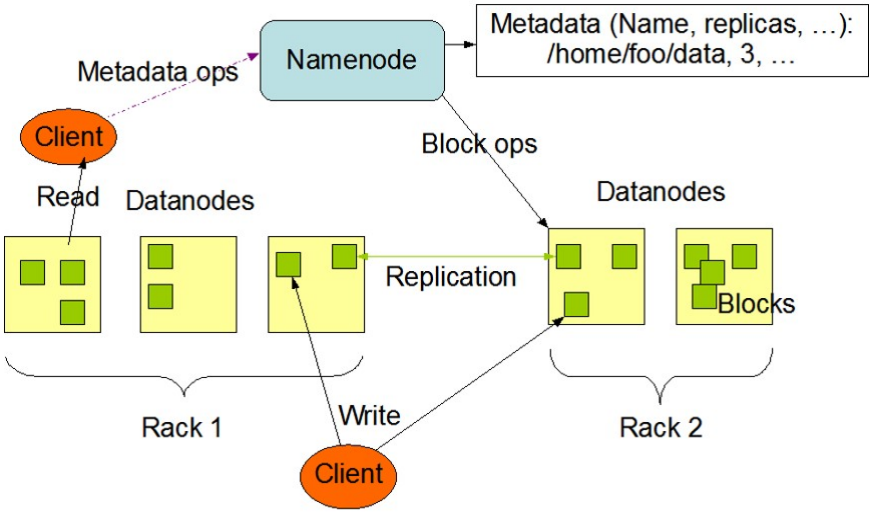


Figure 1: Image showing the HDFS architecture from the HDFS Architecture Guide (Borthakur et al., 2008)

reduce function which aggregates the results from across the DataNodes. As an example, if an Oxford English dictionary was pulled apart and stored across an HDFS file system, each DataNode might store only a few pages. If one wanted to count the number of times the letter "a" appears, a MapReduce job could be used. The Map function would count how many times the letter "a" appears at each DataNode storing its own few pages. The Reduce function would then aggregate the results from each DataNode across the entire dictionary stored on the HDFS. Figure 2 shows how the MapReduce would work.

YARN, which was only added as part of a later version of Hadoop, gives access to alternative ways of processing the data stored on the HDFS e.g. graph or stream processing. In essence, it is responsible for non-MapReduce jobs within the framework. Furthermore, it improves the overall performance of Hadoop by separating out the scheduling and resource management components of the framework from the data processing component (i.e. MapReduce).

In addition, there a great number of related projects and software that exists within the Hadoop ecosystem. Figure 3 gives a visual map to the Hadoop ecosystem. Below is a non-exhaustive list of Hadoop ecosystem software and platforms (Raj & D'Souza, 2019).

- Hive provides a SQL-esque interface for data stored in HDFS.
- HBase is a scalable, distributed database for structured data storage.
- Spark is a compute engine designed with a programming model to support alternative applications e.g. extract, transform and load (ETL), Machine learning and data streaming.
- Mahout is a distributed machine learning library and data mining tool.

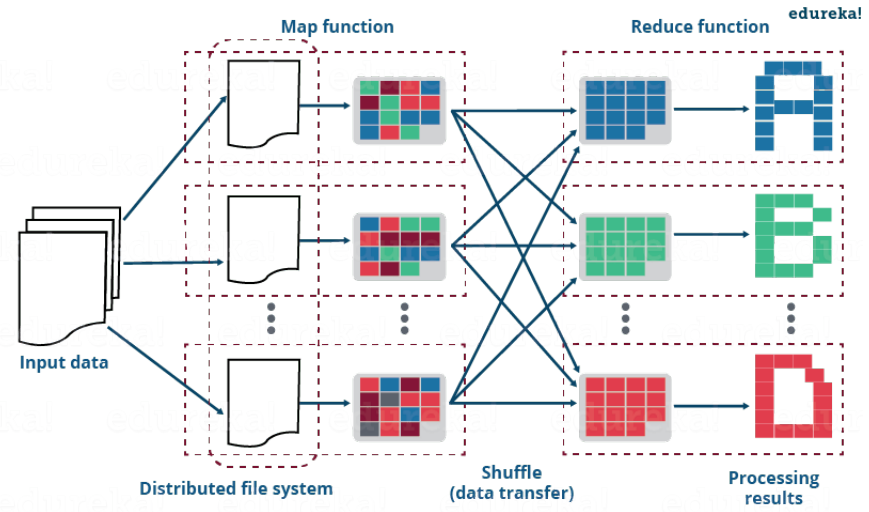


Figure 2: Example of a MapReduce from edureka! site (Bakshi, 2019)

- Arvo is a remote procedure call and data serialization framework.
- Sqoop is a tool for transferring data to and from relational database servers and HDFS.
- Drill is a low latency distributed query engine.
- Thrift provides APIs for client implantation.
- Oozie is a workflow scheduling system.
- Ambari is an open-source administration tool.
- Flumes collects, aggregates and stores large log files to a central location.
- Zookeeper enables synchronization across a cluster.
- HCatalog is a tables storage management tool.

Hadoop forms part of most cloud infrastructure as AWS, Google and Azure all offer Hadoop as a distributed service and it is ideal for solving the processing power problem.

2.4 Apache Spark

Apache Spark represents a step forward from Hadoop. Designed in 2009, it builds on the Hadoop system by offering a higher speed of computation. This is achieved by the extension of the MapReduce functionality to processing in parallel with data held in memory instead of continually reading and writing to the HDFS. Spark does use some elements of the Hadoop framework such

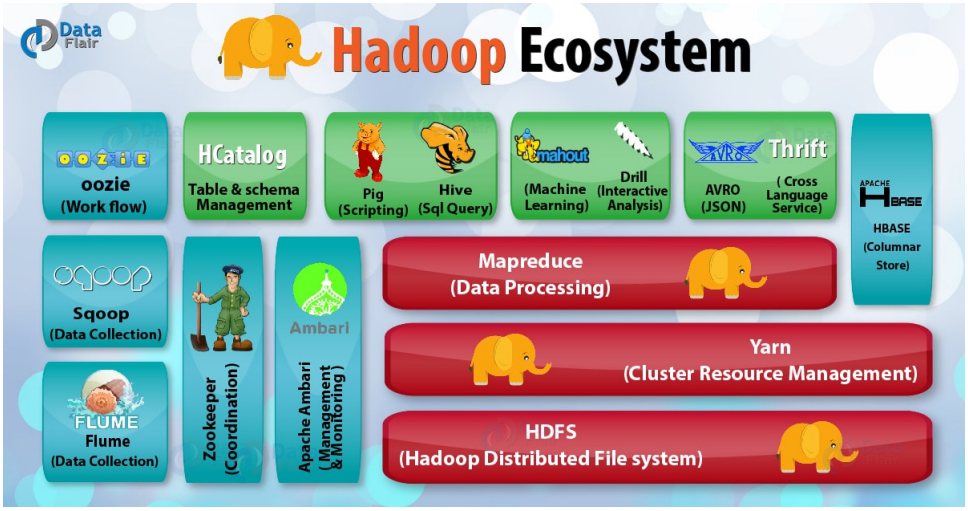


Figure 3: Image showing the elements of the Hadoop Ecosystem from the DataFlair site (Dataflair, 2019)

as the HDFS for storage and the cluster for processing but does its own cluster management. As such, it is often used in conjunction with Hadoop but it can be used as a standalone application to facilitate high-speed computation. It is especially useful as a data analytics engine and for high-speed data such as real-time streaming. As another bonus, it is much easier for a user to learn and use than Hadoop as Spark has developed a host of APIs which work seamlessly with a number of programming languages such as Python, R, and Scala. However, these benefits do come with a higher cost than that of Hadoop (Chambers & Zaharia, 2018).

At the core of Spark lies the RDD - a resilient distributed dataset. An RDD is a collection of elements that can be stored in parallel across the cluster and facilitates the parallel processing that drives Spark's speed. These RDDs can be created in two ways - firstly from the parallelization of existing collections (e.g. data stored on the HDFS) or by referencing an external file or object. RDDs are stored in partitions that are spread across nodes within the cluster. RDDs are stored within partitions across the nodes meaning that data is split and distributed across the nodes. Generally partitioning is done automatically using a default number of partitions (one partition for each 128 MB block of the file) but users can specify how the RDDs are distributed using either hash partitioning or range partitioning (ProjectPro, 2016).

Hash partitioning works by using a key within the file, creating a hash code and using a modulo divide (by the desired number of partitions) to evenly distribute the data across the cluster. Hash functions are largely used in encryption methods or table lookups and essentially map a number to a fixed-sized integer. There are a number of the widely used hash functions as a message digest function (e.g. MD5) or the secure hash algorithm (e.g. SHA-2) (Wang & Yu, 2005). Range partitioning works on the premises that some keys within the data are ordered by a sequence and the keys that are close in the sequence

should be stored near each other or on the same node. Hence range partitioning of the object will use these keys to distribute the data in a way that data with nearby keys will sit together (ProjectPro, 2016).

In addition, Spark works as a lazy executor. There are two types of operations for an RDD - transformations and actions. Transformations are any operation that creates a new dataset from an existing RDD (e.g. map. filter, sample) while actions return non-RDD objects (e.g. take, reduce, collect) (Spark, 2018) Spark will not process any transformations unless an action command is given. This allows optimization of the sequence of commands and adds to the speed at which Spark can process.

2.5 Data Analysis Process

Although there are many variations of the "data analysis process" or "data pipeline", there are 5 main steps involved:

1. Sourcing Data
2. Data pre-processing
3. Data exploration
4. Modelling the data
5. Unpacking the results

The process is presented as a step-wise process but in fact, it is a non-linear, iterative method. Often a practitioner will need to regress in the process to overcome a unforeseen challenge. For instance, in modelling credit default rates, a practitioner may discover that the algorithm is biased against people living in certain neighbourhoods and cities. They would then have to return to the data pre-processing step to either remove or modify the data in a way that removed the bias.

There are many papers and research topics that cover some of these data pre-processing issues and for the most part, solutions can be found in applications such as deep learning and transfer learning, with varying degrees of success. Garcia et al. try attempt to summarise some of the literature around the pre-processing steps in Big Data with a 2016 paper which gives an idea of the scope of the issues and the various approaches to solve the issues (García, Ramírez-Gallego, Luengo, Benítez, & Herrera, 2016).

In practice, the analytical process runs a similar path to the one outlined above but is complicated by many stakeholders involved. The process of solving a problem is begun by either a business stakeholder or an analyst asking a question or series of questions. i.e 'Can we predict revenue for the next quarter?' or 'What is the likelihood of sales continuing upwards next month?' It is then the task of either an individual or team of analysts or data scientists to investigate.

The questions may be of descriptive, predictive or prescriptive nature but will generally follow the process outlined below:

1. Understanding the problem

   This normally is a joint exercise with the data scientist and domain expert where the data scientist can then get a greater understanding of the problem and factors that may cause it. However, if the problem or industry is familiar to the data scientist, this may then just focus on using their knowledge to solve the problem. It is generally a good idea to discuss things like infrastructure, cost and use of the prediction with the business stakeholder or domain expert. For instance, will they want the prediction every month or once off? Is the project likely to go into a production environment? What are the limitations on technology and what is available for the project? etc. This is one of the more overlooked steps in the analytics process but one that can make or break a project.

2. Sourcing the data

   In industry, data is not hard to find. Projects generally start with a clear objective and a large, structured set of data. However, whether the data is complete or relevant to the objective may be debatable. Often big corporates have multitudes of stand alone systems, all with incredibly rich data but all unconnected and hard to relate to each other. Here, many data science practitioners in industry will rely heavily on enterprise data warehouses or data marts which attempt to solve that problem for them. Skills such as database management and querying relational databases are needed. But more and more business are looking at "data enrichment" techniques that combine structured relational database with newer unstructured data. Depending on the problem, data scientists may need to work with database managers and data engineers to source appropriate data and to manage the data pipelines to bring the required data into an appropriate environment for analysis.

   This is especially true when data is unlabelled (i.e. there is no pre-assigned target label) or if there are infrastructure considerations. For instance, if a new set of data resides in a data lake on a cloud platform, a data scientist will need to work closely with both a data engineer to ensure data flow to an appropriate modelling tool as well as a data modeller to ensure that the data is coming through in a structured format.

3. Creating a final dataset

   This step is known as a "Data Exploration" phase. Once all data has been sourced, data needs to be pre-processed. Data is never clean. This data pre-processing step is a catch all phrase for anything such as imputation, feature engineering and synthesis, noise reduction and scaling. It is often an under-appreciated stage of the process but the value of "cleaning" the data should not be underestimated. The next task is to aggregate and summarise the data into a single table format or an input array and squeeze that data for as much information as possible. Machine learning models are exclusively used with a single input of structured featured data rather than a set of tables or relational schema or unstructured data.

In addition, features need to be explored and tested for relevance. It is here that the feature engineering or synthesis process offers the biggest rewards. A quick and easy way to aggregate data from multiple tables in a relational database will save the data scientist/analyst effort and time. Understanding the problem statement, understanding the data and how these fit together will result in a final dataset that is suitable for answering the question asked by the stakeholder.

4. Iteratively brainstorming, testing and adding features to improve the model

   In practice, this is not a process that normally goes smoothly. Understanding the, sometimes multiple, stakeholder requirements, the technical requirement, sourcing data, understanding data and putting together a final dataset is a process that requires time and multiple iterations. Creating features from the dataset to improve performance, represent the problem or add additional information is often a large and time-consuming part of this process. There are often times when projects have altered drastically or been shelved completely due to lack of predictive features or lack of sourced data.

5. Modelling the data

   Once a final dataset is available, the modelling process can begin. This is often another lengthy process that involves running multiple models, hyper tuning and validations. Generally, this will involve its own set of problems in Big Data such as the curse of modularity. Since this is a well represented area of research, this dissertation, while acknowledging it as a non-trivial task, will not cover the machine learning portion of the problem.

6. Unpacking the results

   Part of the process is to try and understand the model or, at the very least, gain some acceptance of the model results from stakeholders. In highly regulated industries, there may be some requirement for understanding and testing the decision making process itself (e.g. the South African Reserve Bank highly regulates credit models for banks). It is good practice for the data scientists themselves to at least test the validity of the model by interrogating the variables that influence the decision. If the main aim of the model is inference or understanding contributory factors, then it is essential to be able to understand how the outcome is related to the features/inputs. In these situations, traditional feature engineering rather than feature representation or feature learning is essential as these features are more intuitive and can be explained to stakeholders in non-technical language.

7. Iterate

   The process described above is not normally one that flows without incident. In some cases, there will be stakeholder management or technologically issues, a new data source to consider etc. In other cases, it may be that the model results are less than required or that there is some unseen

bias in that data. The process generally continues until all stakeholders are moderately happy.

8. Deployment or Measurement

   In some projects, the model is needed in a production environment and once model acceptance has been obtained from the data scientist and stakeholders, the model will generally pass through the software development life cycle. At this point, the data scientist will have some input but the process beyond this point is managed by a team of data engineering and operational specialists. However, it is always a good idea, once deployed, for a data scientist to measure the model regularly and if the results generated do not resemble those within the modelling process, to retrain the model.

Creating a final dataset is often the most time consuming work of the process. And in dealing with Big data, this step poses numerous challenges which can lead to delays. The sheer size and complexity of the data may make normal data pre-processing steps infeasible. Having a framework set up to support this process for Big Data problems will diminish the time taken for the analytics process, decrease the overall time an analyst takes to build a model and be of immense value.

2.6 Feature Engineering

Any model is only as good as the data on which it is trained. Towards Data Science (a popular on-line resource for data science) quotes:

"Data quality might very well be the single most important component of a data pipeline, since, without a level of confidence and reliability in your data, the dashboard and analysis generated from the data is useless." (Gary, 2019).

There exist many definitions for quality data, generally consisting of variations of completeness, consistency, conformity, accuracy, integrity, and timeliness. The most important factor under consideration is that data be fit for purpose. But, as pointed out by Mocnik, Zipf, and Fan (2017), data quality and fitness for purpose may be two different things: a map of an area may contain extremely high-quality data but is not, in fact, fit for a study into soil classification.

In essence, feature engineering and synthesis is a process of squeezing fitness for purpose from a set of good quality data. It is based on the notion that to achieve a good model, a good representation of the input variables/predictors should be made available to the model. This largely means taking the input variables or predictors and transforming or combining them (now called "features") in ways that allow the model/s to extract maximum value and information from them. As most models, and in particular machine learning models, require the input to be in a flat-file structure (normally a data frame or an array with rows as observations and columns as predictor variables), feature engineering is an essential step in the data analytics process when dealing with data stored in multiple tables at different granularities.

Traditional feature engineering is largely considered an informal topic within the machine learning world - it is a largely undocumented and an under-appreciated area of research. Leading data scientist, Pedro Domingos refers to it as a type of "folklore" in Data Science and states that

'This is typically where most of the effort in machine learning project goes. It is often also one of the most interesting arts, where initiation, creativity and "black art" are as important as the technical stuff' (Domingos, 2012)

Feature engineering falls loosely into two main categories: feature selection and feature synthesis. Feature synthesis can then be split into aggregation of data and a transformation of existing data. Each of these serves a different purpose.

In feature selection, the objective is to whittle down the number of features into a few relevant features which allow for simpler models without loss of accuracy or performance. Simpler models are easy to understand and explain and reducing the number of variables makes the learning process less computationally expensive and faster. In other words, it helps reduce the curse of dimensionality (Bellman et al., 1954). Feature selection is sometimes considered as separate from the pre-processing portion of the model. It is the most well-documented and researched portion of feature engineering.

Transformation of data deals with forcing data into a structure that the model can deal with easier. For instance, binning numerical values, converting categorical variables into dummy numerical ones, transforming correlated features and creating new variables that better represent the underlying problem.

As an example of transforming existing raw data into a new feature: The infamous Titanic dataset is a Kaggle entry standard which attempts to predict the survival rate of passengers aboard the ship. Data Camp, an on-line Machine learning training and community site, combines two columns "SibSp" (the number of siblings on board for a passenger) and "Parch" (the number parents or children aboard the Titanic for that passenger) to build a new feature which indicates the size of the family to which the passenger belongs. This improves the overall accuracy of their model on their validation set of data (Hugo, 2018).

Aggregation of data is the expression of raw data in a summary or statistic form for analysis (e.g. sum or average) This concept extends to where data is not in a tabular form but exists within a relational database. This variety of data can be exploited in both small and Big Data.

For instance, if the objective is to predict something about a mobile account user from their call transaction history and each user has multiple call transactions in their history, the first step would be to aggregate the call transaction history up to a user-level using metrics such as sum of call transactions values or number of call transactions.

2.6.1 The Importance of Feature Engineering

Several papers can attest to the importance of feature engineering. In 2016, a paper looking at credit fraud detection strategies reported that the use of their modified periodic features increased overall model performance by 200 percent (Bahnsen, Aouada, Stojanovic, & Ottersten, 2016).

In 2016, Jeff Heaton showed that certain machine learning models respond differently to different types of features. He concluded that the choice of model should influence what type of feature should be engineered. If a model can synthesise the feature on its own, then there is no need for manual feature engineering (Heaton, 2016).

Despite the lack of academic papers on the impact of feature engineering on results, there are multiple references to the importance of feature engineering littered throughout the data scientist community. Andrew Ng - one of the most well-known data scientists is quoted as saying *"Applied machine learning is basically feature engineering"* (Ng, 2013).

In reality, the data pre-processing step, including feature engineering, is exceptionally time-consuming and domain-specific work. In most cases, data scientists, working across industries are given sets of data, sometimes stored in relational databases across a multitude of schemas or domains or data platforms, and have to manually prepare data into a format that is consumable by a machine learning algorithm. Pedro Domingos quotes:

"First-timers are often surprised by how little time in a machine learning project is spent actually doing machine learning. But it makes sense if you consider how time-consuming it is to gather data, integrate it, clean it and pre-process it, and how much trial and error can go into feature design." (Domingos, 2012)

In addition, data can be exceptionally industry or domain-specific. So while a data scientist may understand the way an algorithm will make a prediction, the business owner may have more knowledge around which variables are likely to affect that prediction.

For instance, a study of information systems' (IS) misuse made effective use of graph analysis (or network analysis) to engineer 16 features from over 90 GB of data from four different systems stored on Big Data infrastructure to better understand an individual's daily pattern of IS usage (Lopez & Sartipi, 2018).

Automating the feature synthesis process, or even part of it, is a valuable time-saving exercise for the data scientist. The time saved could be spent in conversations with the domain experts to gain additional understanding or in testing different algorithms and methods.

2.7 Feature Engineering in Big Data

2.7.1 Volume

When data is considered "big" within the volume aspect, the curse of dimensionality is triggered.

The curse of dimensionality as introduced by Richard Bellman in 1954 refers to the fact that as the number of features grows, the size of the training data must grow as well. This means that for a given training set of non-changing size, the predictive ability of the machine learning algorithm diminishes as the number of features increases (Bellman et al., 1954).

When synthesising features on a smaller dataset, there may be a danger of inducing the curse of dimensionality. But when working with Big Data, the size of a training set is not an issue. Only when the number of features rivals the number of instances does the curse of dimensionality becomes a barrier in machine learning on Big Data.

2.7.2 Variety

Data that is large on the variety scale has other issues. In practice, structured and unstructured data often sit in "links". For instance, a standard relational database can be viewed as a linked data with the main customer table which links to additional information about that customer - sometimes at a different aggregation level (e.g. the customer's transaction history). In unstructured data, these links occur as well. Twitter data often contains hyperlinks or retweets which can be viewed as a type of linked data. This often means that data has a high level of correlations and noise in it due to these links. However, this is also valuable information stored in these links. The challenge becomes how to exploit the relationships in these links for feature engineering (Li & Liu, 2017).

2.7.3 Velocity

Velocity in Big Data refers specifically to streaming data. In this case, data processing and feature engineering need to take place in or near real-time. This means feature engineering on a dataset where the number of features is unknown. There are cases where data is sourced, processed and analysed but never stored which makes applying the features engineered from one batch to another unavailable. Examples of this may be real-time spam filters where data is constantly arriving and needs to be classified. Another example is real-time Twitter or Instagram scans for offensive posts.

2.8 Feature Engineering Solutions for Big Data

2.8.1 Representation or Feature Learning

One type of feature representation or learning which is commonly used when dealing with Big Data makes use of neural network algorithms that extract high-level, complex abstractions as data representations through a hierarchical learning process. Deep learning methods can then be thought of as multiple layers of these representation learning methods - each responsible for adding another layer of understanding or feature to the data. In other words, instead of a human using domain knowledge to transform and gather features of a problem, the data is given over to a neural network to perform a supervised or unsupervised learning algorithm which in effect "learns" the features of the data and decides which are useful or not. In this way, the neural network can output a representation of the data fit for purpose. In some cases, this can be a precursor to a traditional regression or classification algorithm. However, in most cases, these representation learning models are bolted on to the front of a neural network which then performs the regression or classification step.

Deep learning is particularly useful for image and speech recognition and does well on high dimensional data as it learns relationships between features. For instance, to try and manually craft features that classify pictures that contained dogs or not, one might want to look at shades of grey within the image. A Local Ternary Pattern (LTP) could be computed on the image - this uses a 3-valued encoding scheme to identify differences in the gray-scale. If the colour a pixel is lighter than some threshold value, then the value is 1, darker -1 and the same, 0. In this way, for each pixel, a numerical value is obtained. This "handcrafted" feature, along with other similarly crafted features looking at RGB colour scales, rotations, shapes, etc. could then be fed into a classifier. (Nanni, Ghidoni, & Brahnam, 2017)

In the case of deep learning, a convolutional neural network with multiple layers could be used instead. The convolutions will learn the features of an image and be able to use these to classify an image.

It might be easy to jump to the conclusion that the deep learning networks are pulling the same features from the image that a handcrafted set of feature might. However a 2017 study states that *"..the experimental results comparing handcrafted features against non-hand crafted features show that the two systems extract different information from the input images."* (Nanni et al., 2017). This paper concluded that the fusion model's performance was higher than using deep learning alone.

It is undeniable that deep learning has a profound effect on the time taken to produce a machine learning model and it *"will have many more successes in the near future because it requires very little engineering by hand, so it can easily take advantage of increases in the amount of available computation and data"* (LeCun, Bengio, & Hinton, 2015).

This method solves the curse of dimensionality and can be used on streamed

data as well, but it still does pose some issues. Firstly, it adds to, rather than diminishes, the processing power issue that Big Data is faced with. Secondly, it struggles to adapt to the concept of linked data i.e. if data is "big" on the variety scale. Thirdly, because it uses a complex neural network, it trades accuracy for interpretability, meaning that the features fed into the classification or regression engine are no longer completely explainable to a decision-maker in business. In addition, deep learning runs up against the curse of high dimensionality which means that it takes a large dataset to train it correctly. Lastly, this still requires a single table input representation of the problem.

2.8.2 Dimensionality Reduction

Dimensionality reduction *"is the transformation of high-dimensional data into a meaningful representation of reduced dimensionality"* (Van Der Maaten, Postma, & Van den Herik, 2009). These new dimensions (or resultant features) are not learned through a neural network which learns which features might be important to the problem. These take all the data and find a way to represent the data in a lower dimensional space. For instance, the popular Principle Component Analysis (PCA) uses an orthogonal transformation to convert possible correlated features into a set of linearly uncorrelated features (Fodor, 2002). Here the first component accounts for the largest portion of variability or information in the data, the second component for the second highest portion and so on. This is an example of a linear, unsupervised, generative and global dimension reduction technique. Any number of these techniques exist and can be both linear and non-linear, supervised or unsupervised, generative or discriminative and global or local. In particular non-linear techniques are sometimes referred to as manifold learning. In this manner, the information contained in an n-dimensional space can be represented in a k-dimensional space (where k < n).

Dimension reduction techniques such as PCA, LDA (linear discriminate analysis) and GDA (generalised discriminate analysis) can be used as a data preprocessing step to clean and organise data before any machine learning algorithm is trained. Each of these techniques has their application and is extremely useful for a different set of problems (Güven, Polat, Kara, & Güneş, 2008) (Li, Zhao, Zhang, & Jiao, 2009).

As a whole, dimension reduction techniques work well to counteract the curse of high dimensionality and are fairly conservative in processing power. However, over a distributed dataset, they run into the curse of modularity. To complete the transformation, the data needs to be held in memory. Plus, when data has the variety aspect of Big Data, these methods fail. However, the biggest drawback of these methods in the business area is their lack of easy interpretability. Outputs from the model must then be converted back into the original dimensions to glean understanding from the modeling process which is an onerous and confusing task. In addition, data must be a single table format for dimension reduction techniques to work.

2.9 Feature Synthesis

The major drawback of traditional, manual feature engineering is that is a time-consuming process that requires domain knowledge. In the majority of cases, the knowledge around the domain and the knowledge required for machine learning, reside in two different heads.

However, as an advantage, traditional feature engineering is applicable to data that is big in volume and variety, and whether in a single file or spread across multiple database relations. And as an added benefit, the features synthesized from this process are completely interpretable and intuitive to a human decision-maker. In other words, nothing prevents traditional feature engineering on Big Data except the amount of effort that is required. However, when this process is automated, the problem becomes one of processing power and dimensionality.

With the rise of "democratized" machine learning (i.e. automated machine learning for non-data scientists) the need to automate the feature learning process becomes important - not only because it is a portion of the actual work but because the data required as an input into the machine learning algorithm is just as important as the algorithm itself.

It was only as recently as 2015 when first inroads into automated feature engineering were made. In a paper entitled "Deep Feature Synthesis: Towards Automation Data Science Endeavours", Kanter and Veeramachaneni (2015) put forward a fully automated "Data Science Machine" (DSM) which included an algorithm that fully automated the feature synthesis process - Deep Feature Synthesis (DFS). It was designed to generate *"features that express a rich feature space"* and more importantly it did so on a relational database structure. Kanter and Veermachaneni seem to have been influenced by the work on generating information from knowledge bases - particularly for use in language semantics. The entity-relationship-entity triple created by Cheng et al. seems to have has a significant impact in the Deep Feature Synthesis (Cheng, Kasneci, Graepel, Stern, & Herbrich, 2011). Despite the focus of the original paper being made on the end-to-end process, the greatest value that emerged was the DFS algorithm.

In 2016, a related piece of work was published. "Cognito", as the system was called, automated feature engineering but only used a single database table. The onus was still on the data scientist to produce a single raw table of data that had been aggregated from multiple tables in the relational database (Khurana, Turaga, Samulowitz, & Parthasrathy, 2016). In June 2017, IBM released a paper on their "One Button Machine" or OneBM which extended the notion of the DFS by allowing feature learning on both structured and unstructured data. In addition, unlike the DFS, this was able to run on an Apache Spark cluster with two machines thus also addressing scalability that is missing from the original DSM. The IBM team in 2018 also overcame the limitation of the DFS from only considering numerical data to including temporal data as well (Lam, Minh, Sinn, Buesser, & Wistuba, 2018).

2.9.1 Tools Available for Automated Feature Synthesis on Distributed Data

Despite the inroads into research in this area, there seem to be only four sources of available "automated feature engineering" tools that could be used on distributed data.

- dotData
- H20.ai
- Xpanse Analytics's
- Featuretools from FeatureLabs

dotData is a Japanese company that was founded in 2018 as a spin-off from the NEC Corporation. Their dotDataPy API on Python for advanced users offers fully automated data science solutions including "AI-Powered Feature Engineering" which is custom built for Big Data as well as completely automated Machine Learning with auto-tuning and proprietary machine learning algorithms (dotdata, 2020).

H2O.ai is the creator of one of the leading open-source machine learning and AI platforms. Their open-sourced platform offers distributed in-memory machine learning with the option of scalability. They make their AutoML library available on the platform but there is no mention of providing an automated feature engineering tool. However, their sister platform H20 Sparkling Water does allow for integration with Spark. In addition, they have a few enterprise solutions for commercial use including Driverless AI - which does include automated feature engineering as well as interoperability of the machine learning model. The platform looks poised for use with Big Data and can source data from HDFS, SQL, Amazon Web Services, Google BigQuery, and Azure (h20.ai, 2020).

Xpanse Analytics is a company founded in Ireland that offers an automated predictive analytics platform including automated feature engineering. They have an in-house solution and one hosted on the Amazon web services platform. In addition, they allow you to see the features and understand what drives the model in plain English (xpanse.ai, 2020).

Last on the list is Max Kanter's spin-off business - FeatureLabs. Off the back of his dissertation, Kanter and Veermachaneni co-founded a start-up after being granted multimillion-dollar funding from the DARPA (the US Defence Advanced Research Projects Agency). It offers 3 main products including Featuretools which is built off of DFS and open-sourced through a Python package. In addition, you can run the Featuretools package through an API (Featuretools Enterprise) on either Apache Spark or Dask. Although the original package is designed to run on a dataset that can be held in memory on a single machine, the guide does give very helpful tips on improving computational performance. It does not seem as if the API is free to use. Their next product is ML Apps - a storefront of ready to use tools for specific uses cases. "Tempo", their

last offering is a hosted platform for building end-to-end machine learning applications using their proprietary automation technology. Both the MLapp storefront and Tempo are commercial undertakings but do take advantage of the Featuretools package (FeatureLabs, 2020).

Unfortunately, the IBM team does not seem to have either a commercial or open-source offering.

There thus does not seem to be a "free to use" version that is ready-to-use for Big Data. A Chinese algorithm engineer provided a version called Featuretools4S which is a version of the Featuretools package that is scalable for Spark - however, this does not seem to be a widely used or fully supported package as the latest release date and update to the package is the day after it's initial release (at the time of writing) (Pan, 2018).

2.10 Deep Feature Synthesis - Explanation

The following section gives a brief explanation of the methodology of the Deep Feature Synthesis Algorithm based on Kanter's initial 2015 paper. To give a practical demonstration of the process, a simple explanatory example with three tables is used. The ERD diagram of the relational database containing the three table is displayed in Figure 4. The tables with values are displayed in Figure 5.

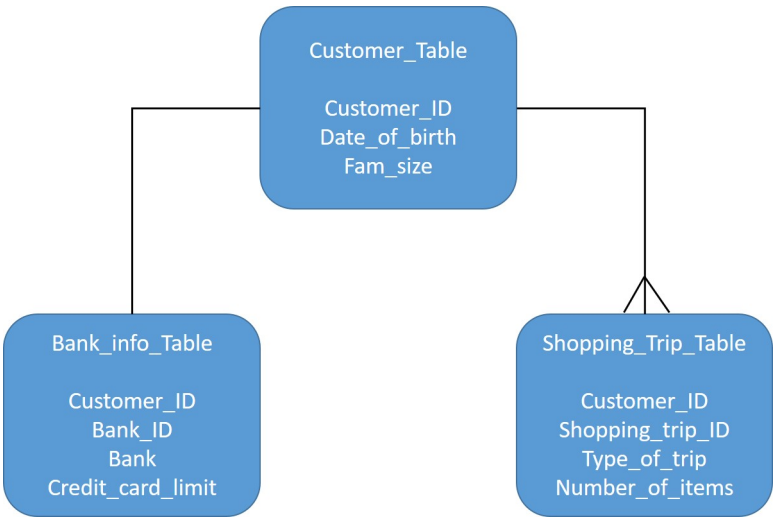


Figure 4: Explanatory Example Pseudo ERD

The DFS algorithm is designed to follow the relationships in the data to a specified base field and then sequentially apply mathematical functions along the relationship path. It works off the basic assumption that there is a set of connected entities and associated tables, and that each table within the entity set should have a unique key, or that an entity can refer to an instance of a related entity by using the related entity's unique key. The input required for the

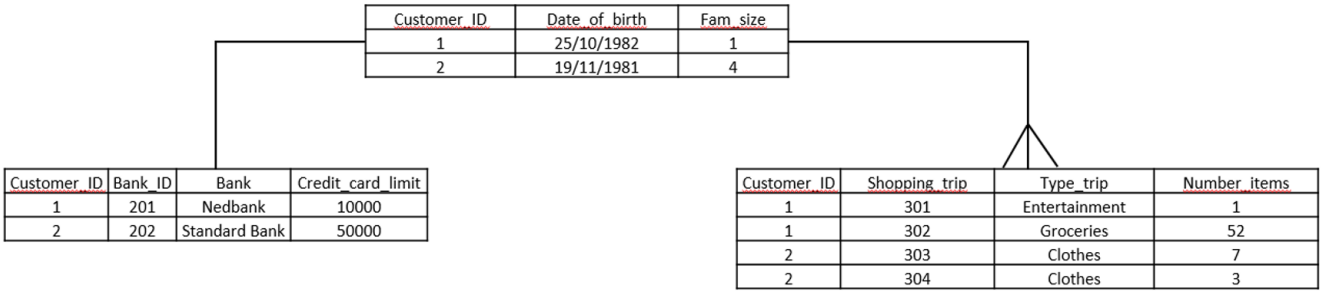


Figure 5: Data in Tables for Explanatory Example

algorithm requires three things:

1. The full entity set
2. A set of relationships between entities
3. The base entity table

In the case of the example, the entity set is the set of the three tables. Each entity (customer, bank and shopping trip) have their unique identifier (Customer ID, Bank ID and Shopping Trip ID). Given the terminology for the DFS, our entity set would be the three tables. The set of relationships would be the customer-shopping trip relationship and the customer-bank relationship. The base entity is the customer table. In other words, the algorithm requires all the tables, the relationships between the tables and the granularity to which it needs to aggregate.

For a given set of entities (denoted $E^{1...K}$) there are 1...J features. Each specific entry is denoted $x^k_{i,j}$ i.e. the value for the feature for i^{th} observation of feature j in the k^{th} entity table. For instance, the first customer's birthday data with be denoted as $x^1_{1,1}$ while the bank name for the second customer is displayed as $x^2_{2,1}$

The algorithm starts by analysing the base entity table and its relationships with other entities' tables by first determining if the relationship is a forward or backward relationship. For instance, if the base entity in our example was the customer table, the algorithm would begin by examining the relationships with the bank table and the shopping trip table.

Forward relationships between two entities' set or tables (E^n and E^k) exist when an instance of i in E^k has an explicit dependency on instance m in E^n. In the explanatory example, an instance of a Bank ID will have an explicit dependence on the instance of a customer in the customer table. Where these forward relationships exist, direct features or `dfeats` are applied. For exam-

ple, applying the algorithm in the explanatory dataset leads to the following `dfeats` group by the instance of a customer:

- Number of unique bank accounts per customer
- Mode of bank accounts per customer
- Count of bank accounts per customer
- Sum of credit card limit
- Mean of credit card limit
- Min of credit card limit
- Max of credit card limit
- Standard deviation of credit card limit

Backwards relationships exist between E^n and E^k where many instance of i in E^k depend on a single instance of m in E^l i.e. a 1-to-many relationship exists between E^n and E^k. The shopping trip entity and customer entity have this relationship where one customer may have participated in many shopping trips. Where these backwards relationships exist, relational features or `rfeats` are generated. These are derived by applying mathematical functions to the collection of features in E^k for instance m in E^n i.e they are aggregation of the multiple observations a single instance of the base entity. In the example, the shopping trip related to individual customers are aggregated up be some means to a customer level.

- Count of shopping trips
- Sum of shopping Trips
- Mode of trip Type
- Min of number of items bought
- Number of unique types of shopping trips
- Standard deviation of number of items bought
- Mean of number of items bought
- Max of number of items bought

Once both the `dfeat` and `rfeat` are synthesised, the algorithm then creates "entity features" or `efeats`. There are the features that can be generated by the base entity table itself. Functions are applied element wise to the $x^n_{i,j}$ entities. These are typically numeric e.g. rounding but also include techniques where a categorical feature is translated to a numerical feature (the numerical equivalent must be decided beforehand). Translation of timestamps (e.g.

Table 1: Features Generated for Explanatory Example

FeatureNumber	**FeatureName**
0	<Feature: Fam_size>
1	<Feature: SUM(bank.Credit_card_limit)>
2	<Feature: STD(bank.Credit_card_limit)>
3	<Feature: MAX(bank.Credit_card_limit)>
4	<Feature: SKEW(bank.Credit_card_limit)>
5	<Feature: MIN(bank.Credit_card_limit)>
6	<Feature: MEAN(bank.Credit_card_limit)>
7	<Feature: COUNT(bank)>
8	<Feature: NUM_UNIQUE(bank.Bank)>
9	<Feature: MODE(bank.Bank)>
10	<Feature: SUM(trips.Number_items)>
11	<Feature: STD(trips.Number_items)>
12	<Feature: MAX(trips.Number_items)>
13	<Feature: SKEW(trips.Number_items)>
14	<Feature: MIN(trips.Number_items)>
15	<Feature: MEAN(trips.Number_items)>
16	<Feature: COUNT(trips)>
17	<Feature: NUM_UNIQUE(trips.Type_trip)>
18	<Feature: MODE(trips.Type_trip)>
19	<Feature: DAY(Date_of_birth)>
20	<Feature: YEAR(Date_of_birth)>
21	<Feature: MONTH(Date_of_birth)>
22	<Feature: WEEKDAY(Date_of_birth)>

weekday, month, year etc.) is also supported. In addition, the feature generation can take into account functions applied to the entire column of features and calculate computations such as percentiles, standard deviation etc.

For instance, observing Table 1 the `efeat` generated off the birthday date features in the customer table, have been broken up into

- Weekday - the day of the week represented by a numeric
- Day - the day portion of the timestamps
- Month - the month portion of the timestamps
- Year - the year portion on the time stamp

Importantly, the `rfeat` and `dfeat` are created before `efeats` are applied as the features of the forward and backwards portion become new columns of the base entity and are included in the generation of `efeat`. Where a related entity has its own relationships, the algorithm becomes recursive. As an example: if the shopping trip entity has a forward relationship with a merchant entity, the DFS algorithm would compute the `dfeat` of the merchant to the shopping trip, then compute `dfeats` of shopping trip to customer before completing the rest of the algorithm. In this way, the algorithm can compute deep features along relationships within the entity set. In this case, the resulting feature would have a depth of two. Built into the algorithm are tracking features to ensure that entity tables are not visited more than once.

2.10.1 Improvement from DFS to Featuretools Package

There has been considerable work done between the original version of DFS which is covered in the 2015 paper and the package Featuretools which is the production version of DFS. For instance: it includes the resolution of the temporal issues which OneBM attempted to solve and the more sophisticated use of "primitives" which allow for a more rich form of aggregation and transformation of features. In addition, it allows users to define their own form of primitives. There is more functionality to the tool than originally existed (e.g. using multiple columns in an entity set to form a feature or specifying the depth to which the algorithm runs). For a broader picture of the full functionality, the reader is referred to the official documentation and Featuretools web site (Featuretools, 2020).

2.10.2 Summary

This chapter has examined in greater detail some of the issues faces around working Big Data, solving for processing power and how the analytics process is approached in industry. In addition, it has had a closer look at feature engineering, its benefits and some of the more common methods of feature engineering on Big Data. It has examined, feature synthesis and outlined how

the DFS algorithm that powers Featuretools works. Chapter 3 sets out the high level details for the experimental infrastructure and the process taken to show viability and effectiveness of automated feature engineering on Big Data.

3 Framework Design

This dissertation aims to show a viable and effective method of automating traditional feature engineering or synthesis at scale. To show the feasibility of this, an experiment was undertaken to demonstrate the steps and assess their effectiveness.

The experiment designed illustrates some of the challenges that an analyst working in industry face, namely transforming data stored in a relational database into a feature matrix appropriate for a machine learning algorithm. The experiment shows that there is a viable solution for Big Data feature synthesis using the cloud computing platform. This scaled experiment uses data across three different tables - each having different but related information in it - and forms a single flat table.

Chapter 2 highlighted a number of options available to automate feature synthesis. The algorithms behind H20, Xpanse and dotData are unavailable on public forums. Thus, in order to conduct the experiment for this dissertation, there are really only two options available - either to use the Featuretools4S as it is scalable on Spark or attempt to use the Featuretools package as it currently stands and leverage some of the thinking that the demos supplied by Featuretools sets out.

Both options were tried and eventually, due to Featuretools4S dependency on an outdated version of Apache Spark on the chosen infrastructure, the option to scale the existing version of Featuretools was followed.

A framework was designed comprising the following high-level steps:

1. Understand the business problem and find the best representation of the available data
2. Set up infrastructure
3. Create a target variable
4. Use hashing on a common key feature to partition the data across the distributed environment
5. Determine an available method or package for automated feature synthesis
6. Create a feature matrix on a single partition of the data, and use this to iteratively explore feature synthesis
7. Scale the solution to parallel processing of all partitions using Spark

This chapter gives a high level overview of these steps while Chapter 4 covers these steps in greater detail. The requirements for infrastructure and the chosen cloud platform are discussed in Section 3.2

3.1 Understanding the Business Problem

The data used for the experiment is a subset of data taken from a bank and has been completely anonymized and masked for customer identification protection before it was received. The donor of the data wishes to remain anonymous. It consists of three raw data tables: customer demographics, fraud incidents, and purchase transactions.

The customer table consists of 98 764 112 customer records recording the age, annual income and if the customer was an individual or a company. The fraud information has 32 204 observations and 9 variables. And the purchase transaction table has 5 314 235 records across 15 variables. The data is taken over a range of time between June 2014 and June 2016. In terms of the Big Data definition, the data would be considered large in terms of volume and as discussed, there could be an argument made for the variety aspect as the customer information system, fraud reporting processes, and transaction systems are generally mutually exclusive in banking systems. However, this data has been taken directly from an RDB and modeled via the Kimball process (Kimball & Caserta, 2004).

The meta data for each of the tables and their variables is represented in Tables 2, 3 and 4 respectively.

Off the back of such data, a likely question would be: is there is any indication in a customer transaction history or demographics that could be a predictor for fraud on a customer's account? For instance, was there more activity than usual on accounts where a customer has reported fraud? Were there specific merchants that are common to the reported fraud cases? Are customers over a certain age or with an income group more likely to report fraud on any of their accounts? For the sake of the experiment, the underpinned question will be: Can we predict the likelihood of fraud occurring on a customer's account based on their recent transaction history? As a reminder, the aim of this dissertation is not to solve the question but to illustrate how the feature synthesis process could take place in a viable and effective manner. However, it is useful to do so with a particular question in mind.

Generally, more information would be available to a data scientist working in industry and there would be stakeholder meetings to ascertain details around how and why the business would use this information. However, as this dissertation is less interested in the actual business questions and seeks rather to show the feature synthesis and data pre-processing steps that would be undertaken, a broad business question will suffice.

3.2 Framework Infrastructure

This dissertation's main objective is to show that automated feature synthesis is viable on datasets that are big in the volume aspect. But in order to do so, the issues of processing power and curse of modularity need to be overcome first - in other words, there is a need for an infrastructure that can handle the

Table 2: MetaData for Customer table

Variable Name	Type	Description
cst_id	varchar(50)	Unique customer identifier
age	varchar(50)	Age of customer
anul_grs_incm	varchar(50)	Annual gross income
hogan_cst_tp_desc	varchar(50)	Type of customer e.g. individual or commercial

Table 3: MetaData for Fraud table

Variable Name	Type	Description
cst_id	nvarchar(max)	Unique customer identifier
cc_frd_id	nvarchar(max)	Unique fraud identifier
cc_frd_id_2	nvarchar(max)	Unique fraud identifier
invstgr_no	nvarchar(max)	Employee number of investigation
src_crt_tms	date time	Fraud case creation date
ls_incdt_tms	date time	Fraud incident date
frd_eff_dt	date time	Fraud effective date
frd_st_dt	date time	Date of last fraud case status change
cams_frd_st_tp_code	nvarchar(255)	Fraud status code
cams_frd_st_tp_desc	nvarchar(255)	Fraud status description

storage, processing and machine learning of Big Data.

3.2.1 Infrastructure Requirements

To overcome the stated problems of processing power and modularity, the following requirements for an infrastructure needed to be met:

- **Data Storage** The infrastructure needed to be able to meet Big Data storage requirements preferably in a relational database or similar. This is to replicate how most structured data is stored in both small companies and large corporates. Although many cloud platforms allow for streaming and/or unstructured data in data lakes or the like, it is important to show how automated feature synthesis can apply to the largest proportion of an enterprise's data.
- **Integrate with Python packages and Big Data frameworks e.g. Spark** Most machine learning requires the use of various Python packages and, for Big Data, also of Apache Spark and so the infrastructure must be able to integrate with Spark and in some way, allow for scalability. In addition, the infrastructure used must resolve some of the limitations of the Featuretools package (namely that the data must be held in memory and is dependent on data stored in Pandas DataFrames) by leveraging off the Spark infrastructure.

Table 4: MetaData for Purchase Transaction table

Variable Name	Type	Description
tm_prd_id	varchar(50)	Unique time index of transaction
cc_ar_id	varchar(50)	Unique index for card account
cst_id	varchar(50)	Unique index for customer
crd_id	varchar(50)	Unique index for card
txn_no	varchar(50)	Unique index for transaction
txn_date_con	date time	Date that the transaction took place
date_rcvd	date time	Date the transaction was received by bank
txn_amt_orig_ccy	float	Original transaction amount
txn_amt	float	Cleared transaction amount
mrcht_no	varchar(50)	Unique identifier of merchant
mrcht_nm	varchar(50)	Merchant Name
mrcht_cty_code	varchar(50)	Code for city in which merchant resided
cams_mrcht_cgy_tp_code	varchar(50)	Code indicated to which industry merchant belongs
cams_mrcht_cgy_tp_desc	varchar(50)	Description of industry to which merchant belongs
mrcgt_str_adr	varchar(60)	Merchant street address
mrcht_city_nm	varchar(50)	City in which merchant resided

- **Support and Community** It is preferable if there is a community or support structure for working in the cloud platform. On-line support, guides and documentation are especially important when working with Big Data.
- **Affordable** Because most cloud computing platforms charge on a pay as use basis and because the size and variety of data, and number of different components makes the system complex, the infrastructure needs to be fairly affordable to set up, run and maintain.

For this work, it was decided to set up the infrastructure for the experiment on AWS because there was a big community of users to reach out for support when issues are encountered. In addition, it represented a set of skills that would be used in enterprise at future dates.

The design of the infrastructure solution(see Figure 6) is based on that of a Featuretools example (Koehrsen, 2018). A local instance of Jupyter notebook uses a soft shell (SSH) interface into a Spark cluster running on three instances of EC2 and accesses data from an S3 bucket.

Figure 7 is a representation of the infrastructure that has been set up for the experiment. Below, each component is fully explained.

3.2.2 Privacy

A virtual private cloud (VPC) was used in order to be able to work with Big Data distributed over a network in a way that closely resembles the traditional

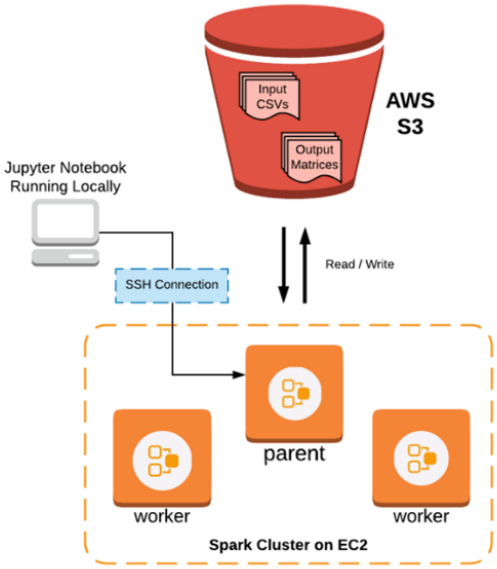


Figure 6: Featuretools Demo Architecture (Koehrsen, 2018)

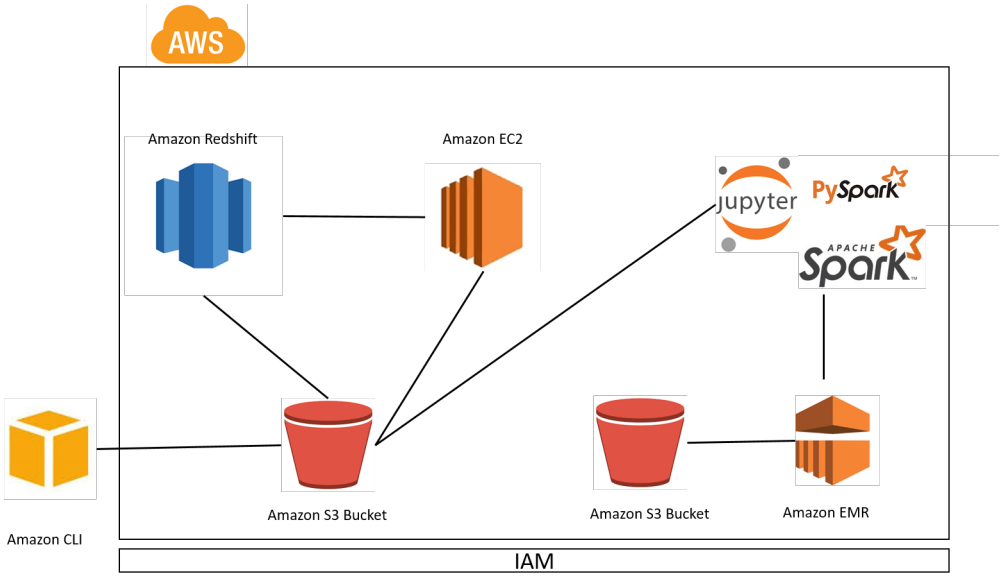


Figure 7: Infrastructure for Experiment

manner. With a private cloud, the environment is housed on dedicated infrastructure to which only the "owner" has access. A VPC functions as if it were a private cloud but the environment sits as a private network within the public domain. Within this private network on the cloud, you can control user access, IP access and security. This is managed through subnets, internet gateways (IGW), route tables and security groups, etc. This gives the users of a VPC the best of both worlds - security and access control while still having access to easily scalable infrastructure on a pay-as-use basis making it less expensive and more flexible. AWS has a version of a VPC as part of their offering.

Despite these advantages, using a VPC in the AWS environment can lead to some complications and extra work in making sure that each component of the infrastructure is able to exist with the VPC and communicate with other components so that the VPC operates as a whole. For instance, the EMR instances and S3 instance must be on the same VPC to be able to load data into the EMR. In addition, ports for Jupyter notebook instances etc. must be manually added to the security groups in-traffic rules. However, the benefits of a section private space outweigh the complications especially when working with enterprise data.

3.2.3 Access Control

Most IT platforms - cloud or not, public or not - have a basic requirement of restricting or granting access to the system. In a large, multi-component system it is generally beneficial to be able to define roles and access permissions as jobs necessitate. In this way a new business user can be prevented from deleting or creating infrastructure on a VPC. The AWS Identity and Access Management (IAM) service provides the ability to manage access to your services and resources by allowing the creation and management of users and user groups and permissions for these groups.

In this work, two groups were created and a single user assigned to each.

- **Administrators** This group's objective is the setup and maintenance of the VPC and resources as well as creating new users and access groups and assigning new users to groups etc. There is only one user in the group which is the main AWS root access account.
- **Users** or "Masters-Users" was created to give users access to the infrastructure but not to allow them to change the setup of the resources. Currently, there is only one user in this group.

It is good practice to separate out the "build" and the "use" access. It lends a degree of sustainability and robustness to the infrastructure. If in the future, others need to access the infrastructure, there is already a solution and the only action needed is to add a user with permissions as required. Although for the majority of the experiment the administrators' user account was used.

3.2.4 Data Upload and Storage

The infrastructure depicted in Figure 7 shows that AWS Command Line Interface (CLI) is used to upload data into the cloud.

The CLI is a unified management tool that allows users to control multiple AWS services from a central location through a command line interface. It allows users to manage services through scripting which makes it possible to automate the setup, running and dismantling of AWS resources and services which eventually saves time for the user who is comfortable with shell scripts.

In this work, it is used as a data uploading tool rather than a management tool.

The AWS Simple Storage Service (S3) buckets are object storage solutions offered by AWS. They are a place to store data but they can hold more than simple csv or txt files - they hold scripts, files, metadata, unstructured data, etc. They can be thought of as a "staging" or "hold-all" area within the cloud. They can be used to do back-up and storage, create archives and act as data lakes. They are also the easiest way to upload or export items on or off the cloud. The infrastructure depicted above utilises two S3 buckets.

- **Upload bucket** The first bucket is used to hold the structured data for the experiment ready for upload into the Redshift database. Files are loaded as csv files but it is important to note that one can only load files that are less than 160 MB through the S3 buckets interface. For files that are bigger than this, the AWS Command Line Interface must be used. In addition, this bucket also holds the partitioning files for the EC2 output in which the data is parcelled for upload into the EMR cluster. In addition, it also contains the final feature matrix that is generated as the final output.
- **EMR bucket** This second bucket stores all the information and data for the EMR. This includes setup files and information logs from the EMR cluster. For instance, when setting up the EMR, a shell script can be used to "bootstrap" actions across the nodes. In addition, this bucket is used to hold the logs of Spark jobs and to store Jupyter notebook scripts.

AWS has a number of different options to store data in a database including NoSQL databases such as key-value, document, graph, and time-series databases. In terms of relational databases, two alternatives were considered:

- **RDS** offers a managed relational database much as you would see in any enterprise. However, it offers a choice of six database engines to power the database: MySQL, MariaDB, PostgreSQL, AWS Aurora, Oracle Database, and SQL Server. The choice of engine would depend on the needs or familiarity of the business but the choice of engine combined with the scalability of the AWS make an RDS a powerful option. However, RDS sits on top of virtualized servers and although scaling these servers is an easy exercise, it does mean that data is not stored in a distributed manner. In addition, it caters for on-line transactional processing (OLTP) through the Aurora engine.

- **Redshift** The Redshift database is primarily designed for the storing and querying of Big Data into the petabyte range through a cluster node architecture. Unlike RDS, it only offers a PostgreSQL engine but does allow the user to specify the number and size of nodes. In addition, it allows users to dynamically resize their Redshift instance through the "elastic resize" feature. Redshift does not cater for an OLTP engine and it is not recommended to be used as such. But it is the most popular of the AWS databases due to its slightly cheaper pricing options, faster performance, and scalability.

As the purpose of this dissertation is to deal specifically with automated feature engineering on Big Data, it is a natural choice that a database designed for Big Data would be chosen. Hence, a Redshift database was chosen.

The database used is a Redshift database with 3 nodes of dc.large node size (2 vCPU, 15 GB of memory with 0.16TB of storage and an I/O rate of 0.6 GC/sec). The cost for each node is $0.25 an hour - which means that for three nodes, this Redshift cluster costs under a dollar for an hour. However, this option falls under the free tier and was not charged.

3.2.5 Data Processing

Data processing was facilitated at two points within the infrastructure using two different AWS processing tools - an AWS Elastic Compute Cloud (EC2) instance and an AWS Elastic MapReduce (EMR) with Apache Spark.

The EC2 offering from AWS is possibly the most used service from AWS. Basically, it provides "virtual computers" that exist within the AWS infrastructure. These "instances" can be spun up or dismantled as needs arise and change - the user can specify the CPU, storage, operation systems, etc. to their own specification or copy pre-made instance types. Security and access can be managed through key pairs, the IAM and security groups and AWS gives a host of ways to interact with the instance - e.g. access through an SSH port or the ability to remotely connect to the EC2 instance. In essence, a user can "rent" a computer/server of their own specification for as long or short as they need it.

MapReduce processing was achieved by setting up AWS EMR (Elastic MapReduce) using Apache Spark and a cluster of AWS EC2 (Elastic Compute Cloud) instances accessed through a Jupyter Notebook with PySpark kernel. The charge for the use of an EMR cluster is on pay per instance per second (i.e. you pay for each second that an instance is running) and the services allow dynamic resizing as needs change. The setup is customizable for customer needs and as from the 5.14 release, users can easily access the power of the EMR through the Jupyter Labs notebooks which are now supported seamlessly on the EMR cluster. In addition, the EMR cluster and Jupyter Labs set up supports the use of PySpark - the Python API for Spark.

Setup steps and screenshots for the EC2 and EMR instances can be found in Appendix A.

3.3 Create Target Labels

The input data lacked a label for each customer which represents if they have reported a case of fraud or not. The fraud table contains a list of customers that have reported fraud but it is not an exhaustive list. By combining the fraud and customer tables, a target variable can be created. In addition, it will be useful to record for which data points the fraud was reported.

3.4 Partitioning of Data

In order to understand why partitioning is necessary, it is important to understand how the EMR cluster will store and process the data.

Data in a distributed data store means that data is not located in a single place. The Hadoop infrastructure that the EMR cluster runs will partition the data across the available servers in the cluster. In the case of data sitting across multiple tables, this opens up the danger that a customer's information may be sitting in different places. This poses a problem for the envisioned feature synthesis across tables. If the aim is to combine data using a function that takes advantage of the MapReduce paradigm, then it is necessary for all data, across the three tables, to be sitting in the same location within the cluster. In addition, the DFS algorithm has a reliance on Pandas DataFrames which means that data cannot be stored in an RDD on the HDFS.

In order to avoid these two issues, data is partitioned manually using an MD5 function and stored in individual files on the AWS S3 bucket. Essentially, the MD5 will convert a string to a 128-bit value. When used on the three tables, all the unique customer identifiers will be converted to the same value and the data can be parcelled according to these values and written out as partitions. In other words, it ensures that all the data in a single partition relates to a certain range of customer identifiers. This replicates the file structure that would have occurred if the data was stored in an RDD using a hash partitioning.

3.5 Feature Engineering on a Single Partition

In order to first master exactly which functions are needed for the preprocessing and feature synthesis, a single partition of data was initially used. This allowed for the complexities and details to be understood on a small scale that could then be scaled. During the experiment, this data was read out of the infrastructure and a local version of Jupyter notebook was used to process the single partition of data. This could have easily been accomplished on the EC2 instance but was cheaper to do on a local instance. In this step, the Featuretools library was used extensively. The defined functions were perfected and the infrastructure was spun back up in order to complete the experiment. This approach meant the cost of resources in the cloud environment was limited and gave the analyst the time to explore the optimal solution. This showed

how versatile and cost-effective the dynamic environment of a cloud computing environment can be.

3.6 Scale the Solution

Once the final cleaning and synthesis functions have been defined, the solution can be scaled for the entire Big Dataset. This involves utilising the parallelization feature of Spark to read in the data and push the data pre-processing and feature synthesis functions to each set of partitioned data. Because all records pertaining to a single customer are in the same partition, the function is able to successfully execute.

The last step is to read in and combine all the feature matrices and output a single matrix. The same is done to the files containing the labels which produces a consolidated list of target variables. .

3.7 Data Flow through the Infrastructure

In this section, the data pipeline is discussed to describe the flow of data end-to-end and how the different components interact together.

The first step is to read data into the AWS S3 bucket through the AWS CLI. Table structures in the Redshift database are prepared and the data is copied into them. This step may seem superfluous - why should the data be stored in a database when the next step would be to read it into an EC2 instance? Why not bypass the Redshift database altogether and upload directly into an EC2 instance? In reality, the storage of the data in Redshift would be excluded. In industry, this data would already be housed in some database to which the analyst would have access. However, in this case, the data instead resides in csv files and the Redshift database symbolises the starting point of the data flow with which an industry analyst would be faced.

In addition, this allows initial SQL queries to be run to understand various aspects of the data such as data quality, volumes, missing data, etc. At this stage, some data cleaning takes place in order for the data to better represent the business problem. More details of this ate given in Chapter 4.

On the EC2 instance, a local copy of a Jupyter Notebook is set up and data is read into the EC2 instance using a database driver library called psycopg2. Tables become available for processing, visualisation, etc. and results can be written back to the database or into an S3 bucket using a popular package called 's3fs'. The EC2 instance may represent a blocker or funnel point for processing e.g. if the EC2 instance is not sufficiently large to handle the size of the data. However, the flexibility of the cloud means that the EC2 instance can be scaled and set up for any amount of data and processing power required. The output from the EC2 cluster is a set of partitioned data files that sit on the S3 bucket.

The data is then accessed by the EMR cluster from the S3 bucket. The cleaning and pre-processing, and feature creation functions are processed in parallel on the partitioned tables. Each partition then contains its own feature matrix which contains the features for its data. The final feature tables from each partition are then read in, combined and written out to an S3 bucket. At this point, the data becomes available for possible next steps i.e. machine learning, further analysis etc.

As the last step in the experiment, the entire pipeline was run from start to finish. The time it took to run was then noted.

3.8 Summary

A framework was designed for automated feature synthesis on Big Data in a distributed environment. The steps required are: understand the problem and obtain the best raw data representation accordingly; set up cloud infrastructure; create the target variable for supervised learning; hash on a common key so as to partition data in a way that avoids the curse of dimensionality; explore feature synthesis on a single partition; and finally scale the solution to parallel processing on all partitions using Spark and MapReduce. Using AWS, privacy and access is controlled through VPC and IAM, data storage facilitated by using separate S3 buckets for initial load and EMR processing, and a Redshift database is employed which can be dynamically resized when necessary.

4 Implementation

This chapter will describe how the steps detailed in Chapter 3 were implemented so as to set up the infrastructure and produce a full feature matrix. The focus will again be on how can this be done as opposed to producing the best feature matrix that can answer the example business question.

4.1 Understanding the Business Problem

The first step in an experiment such as this, is gaining a more in-depth understanding of the data. This is important to gauge how representative the data is of the problem that needs to be solved. For the example used in this experiment, that question is: Can we predict the likelihood of fraud occurring on a customer's account based on their recent transaction history?

4.1.1 Understanding Relationships

Understanding how these three tables are related to each other is essential if the aim is to amalgamate them into a single table. Figure 8 shows a ERD diagram for the tables. The ERD shows a one-to-many relationship between the customer and fraud relations, and a one-to-many relationship between the customer and purchase transaction tables. i.e. a single customer can have many transactions and a customer can have multiple fraud cases. However, there is no direct connection between the fraud and transaction table i.e. neither table has the primary key of the other table as one of its columns.

In other words: it is not possible to connect a fraud case to a specific transaction. The only link that connects the fraud and transaction tables is the customer. This makes the customer dimension an obvious choice for granularity or target entity for the experiment.

In addition, looking at the information, there are account and card identifiers in the customer table as secondary keys but no fact information for them - suggesting that this information was not shared. A customer may have multiple accounts with multiple cards connected to those accounts but this information is unavailable for the experiment. Likewise, there is no way to connect a fraud case to a specific purchase transaction, card or account - only to customer.

On examination of the business problem statement (predict the likelihood of fraud occurring on a customer's account based on their recent transaction history), there is insufficient information to accurately answer. A modification to the business statement would be: Can we predict the likelihood of fraud occurring to a customer based on their recent transaction history and demographic information? There is a subtle change to the question which is better supported by our data.

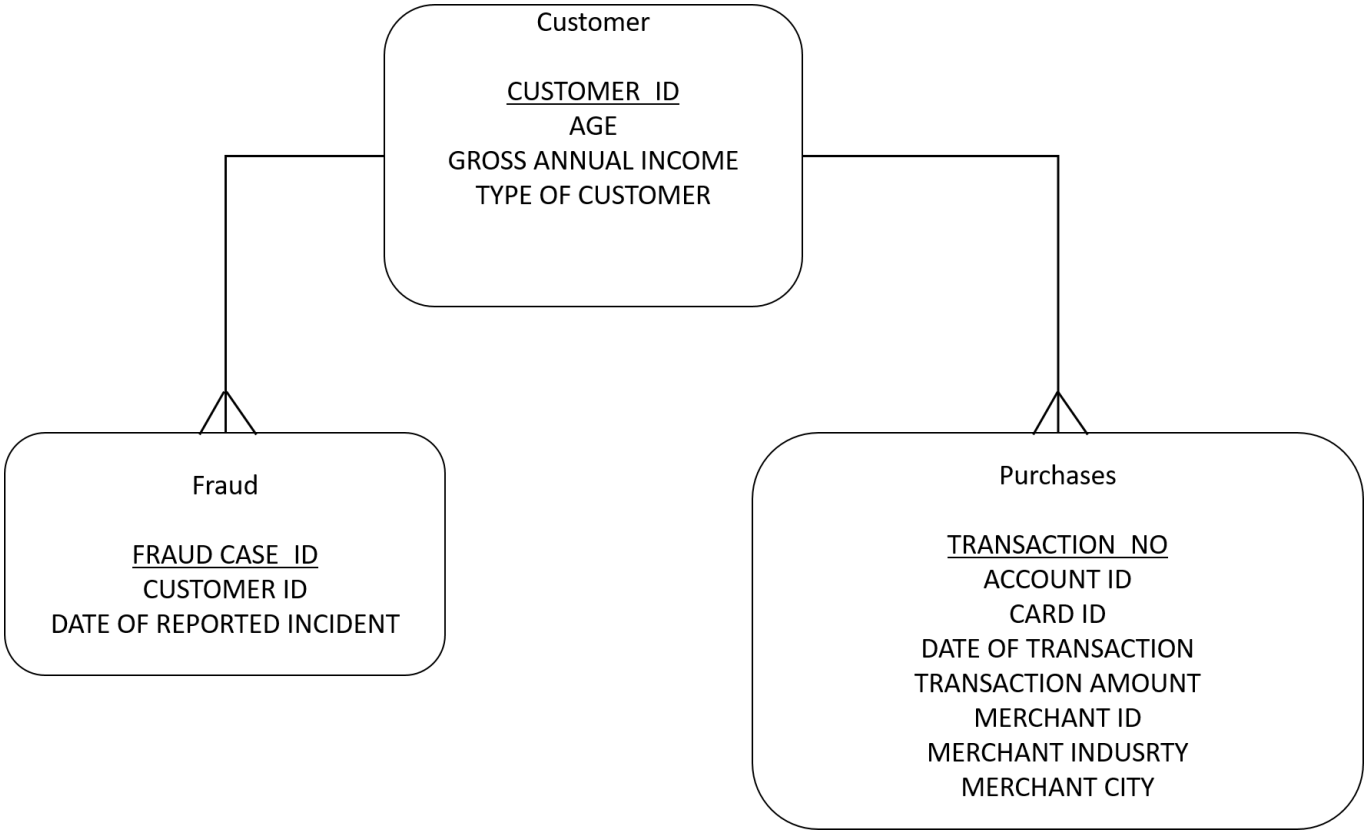


Figure 8: ERD Diagram showing Relevant Data for Experiment

4.1.2 Checking Unique Identifiers

Because this data has been "white-labeled" and masked, a prudent step is to check for duplicates and ensure that any unique identifiers are intact. Although this may seem unnecessary, it is a good fail safe check that will save time later, as even in a governed data environment, mistakes can happen.

When explored, there were approximately 3.2 million duplicated records in the customer table, 260 duplicate fraud records and 785 thousand in the purchases table. Further, the transaction purchase table contained null values for the unique customer identifier and in addition its unique identifier was not, in fact, unique. In the customer and fraud tables, duplicate observations were removed. In the purchase table, the observations with null customer identifier values were removed and a new transaction identifier was made for each transaction to replace the non-unique transaction identifier.

4.1.3 Understanding Timing Issues

On initial data exploration, the timing difference in the data was noted. Fraud cases information ranges from June 2014 to June 2016 but the purchase data only ranges from December 2015 to February 2016. Figure 9 shows this. If the question tackled in the experiment is to identify if purchases transactions impact fraud cases on a customer level, then the information that is available in the fraud table must match the timing of the purchases. i.e the fraud must occur during or after the transactions. As a consequence, the experiment only considers information from the Fraud table between 1st December 2015 and 7th March 2016 (the additional few days to accommodate for if end February transactions contributed to March fraud cases)

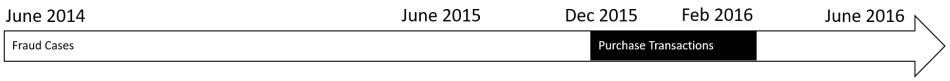


Figure 9: Timeline Difference in Data

4.1.4 Multiple Fraud Cases

During the initial data explorations, it was noted that there existed some anomalies in the Fraud data, namely that there were multiple cases of fraud per customer during the experimental period.

It seems reasonable that a customer may have recorded two instances of fraud within a given period. However, since there is no way to isolate cases of fraud to specific transactions, only the latest fraud case per customer is included in the experiment. In other words: a customer reporting any case of fraud would have all of their purchase transactions examined.

4.1.5 Customers with No Transaction History

During the data exploration, it was noted that the large majority of customer records did not have any transaction history for the period. This is probably because the customer table represents *all* customers that have existed in the bank regardless of product and status of accounts. However, the business problem being "predict the likelihood of fraud occurring on a customer's account based on their recent transaction history and customer demographics" implies the customers should be active i.e had at least one transaction within the 3 month period for which purchase information exists. A similar analysis was taken on the now restricted Fraud table - there were fraud cases were no activity had occurred on the purchases table. Fraud *may* have occurred as a result of inactivity (e.g. stolen card or identity theft etc.) but, once again the problem statement is only concerned with where fraud occurs as a result of recent activity on behalf of the customer. Due to these reasons, the customer and fraud data were limited to only include those customers that had at least one transaction in the purchases table.

In summary, to better prepare the data to represent the business problem, the following steps were taken:

1. Relationships between tables were examined and the level or grain of the problem amended.
2. Tables were de-duped for unique indices and duplicates removed.
3. Null values of the customer unique identifier were removed from the purchase transaction table.
4. A new unique identifier for the purchase transaction table was created.
5. Timelines of the data were examined and the Fraud table restricted.
6. The Fraud table was further restricted to include a single fraud case per customer, namely their most recent one.
7. The Fraud and Customer tables were restricted to only reflect customers that had been active in making purchases.

These steps limited the size of the customer and fraud data to 3380 fraud records and 2 023 036 customer records. However, it must be remembered that the data is now a better representation of the business question.

Appendix D contains the SQL script that executes these steps.

4.2 Setting up the Infrastructure

Chapter 3 gave a high-level view of the technology stack that was set up and used for the experiment. Appendix A shows some of the step-up screens. This section gives more detail on the setup steps.

4.2.1 Uploading Data to AWS S3 Bucket from AWS CLI

Initially, it was envisioned that the raw csv files could be uploaded directly through the S3 bucket interface. However, there is a restriction on the size of the data that interface allows - 160 MB is the biggest file size allowed for a direct upload. For larger files, the AWS CLI is a better option. The AWS CLI is fairly easy to set up (step by step instructions can be found in the AWS help files) but the user must know their AWS credentials (the AWS Key ID and Secret Access Key) which can be found in the IAM user security credentials page for the created user. These details must then be entered via the `aws configure` command in the command prompt window. It is important to enter these credentials correctly and precisely - the lack of both these elements will cause multiple hours of misleading research concerning error messages. It is suggested to change the font in the credentials file to see the difference between lower case letter "l" and the upper-case letter "I". Once uploaded the

tables exist as raw csv files in the S3 bucket. The upload command must reflect both the S3 bucket and destination file, and the file location and name of data in the user's local directory.

Screenshots of the upload are found in Appendix B.

4.2.2 Uploading Data from AWS S3 bucket to AWS Redshift

To upload the data into Redshift, the user must create the schema and the physical table structures before using the 'COPY' command to shift data from the S3 bucket location to Redshift. As with the upload from AWS CLI, AWS credentials for the user are needed. In addition, it is useful to note that if you are copying from a csv file, this file type and delimiter should be specified as well as the header row if applicable. Not doing so will cause errors in the script. There are several options that a user can specify to customize their upload (such as `ignoreblanks` and `fillrecord`). However, if a user specifies a csv file in their upload, some of these will be unavailable. The AWS documentation is extremely helpful. The table creation and upload script can be found in Appendix C.

4.2.3 Setup and Access of EC2

The EC2 instance setup was part of the free tier and was set up using the t2.microinstance type which has 1 vCPUs, 2.5 GHz and 1 GB of memory which runs a Windows OS. The main reason this instance was chosen was the price. However, the range of pre-packages goes all the way to an "i3en.metal" which has 96 vCPUs, 3.1 GHz and 768 GB of memory. A secondary reason was that after our data cleaning exercise, the size of the instance was thought to be appropriate. However, during the full run of the experiment, it was found to be too small and had to be dynamically resized.

There are two main ways of accessing the EC2 instance: using an SSH terminal to execute actions or using an RDP (a Remote Desktop Protocol). To use either of these options, the user must ensure that Port 22 or 3389, respectively, are open on the security group that governs the EC2. The IP address can be specified for access or be left as open which will allow all computers to access the EC2.

In addition, using either option to access the EC2, a keypair file is required. You can generate the keypair file from the "Network and Security" tab on the EC2 screen. It is important to keep this keypair for future use as the user will not be able to access the EC2 without it.

In this setup, the EC2 remote desktop was used and the EC2 instance operated as a "normal computer". Additional set up was required - Anaconda (including Python and Jupyter Notebook) was installed on the EC2 and an additional port was opened. The additional port (5439) allowed a direct connection from the Redshift database to the EC2 while the next steps (partitioning) took place

in Python. In addition, the AWS CLI was set up and configured to allow the Python package `s3fs` access. The package is an interface that allows the handling of S3 buckets in the same way as file directories

The data was read directly into Python from the Redshift database using the psycopg2 library which is a PostgresSQL adapter engine that allows the Python access to the SQL database.

A view of all open ports on the EC2 instance and the Python script (including the connection details to Redshfit) can be found in Appendix A, and E respectively.

4.2.4 Setup and Access of EMR cluster

The EMR cluster was set up with EMR version 5.27 with three nodes of 4 vCore, 16 GB memory and 64 GB storage which was made up of 1 Master and 2 Worker nodes with the following software installed: Hadoop v.2.8.5, JupyterHub v.1.0.0, Spark v.2.4.4, Sqoop v.1.4.7, Livy v.0.6.0 and Mahout v.0.13.0. When JupterHub is selected as an option in the setup the port 8888, which allows the Jupyter Labs connection, is automatically opened on the security group specified for the EMR cluster.

During setup there is an option to 'Bootstrap' any software or packages that a user may need across the cluster i.e. after setup the user will not need to install software or packages on each of the nodes separately. In the case of the dissertation experiment, the Spark *sc.installpackage* function was used to install necessary packages on the EMR cluster. This meant that packages etc could be installed as the need arose during the experiment rather than having a full list of packages required during setup.

There is an option to use soft-shell commands (SSH) into the EMR - as with the EC2 instance. The required port (22) must be open to do so. However, the Notebook functionality which allows a JupyterLab interface into the EMR cluster is extremely useful and was used as the main access into the cluster. The notebook interface allows users to select different kernels (e.g. PySpark, Python or R) and also allows the user to switch between different exiting EMR clusters without changing their notebook or scripts. In addition, logs of scripts and jobs are stored automatically in the S3 log bucket for the EMR cluster. This means that analysts have the option of switching the cluster used to process their jobs without losing any time or effort in changing or copying their scripting or workbench environment.

4.3 Creating Labels for Fraud Cases

The label creation code sits within the same script as partitioning the data.

The first step in the script creates a label to indicate if a customer has reported fraud on their account and at which date that fraud was reported. It begins

with a full list of unique customer identifiers (taken from the customer table) and left joins this to the fraud table. The only variable that is used from the Fraud table is `ls_incdt_tms` which represents the incident time. It was decided that any other variables from the table would only be known *after* the fraud incident (e.g. who the investigator was etc.) These details may be of value if the analysis was exploratory or descriptive in nature but not for machine learning that involves prediction, as in the customer fraud example.

If a `ls_incdt_tms` is recorded then an indicator for fraud (named `frd_flag`) in the customer record is updated to a yes('Y'). If `ls_incdt_tms` is a null value after the left-join, then the script firstly, updates the `frd_flag` to a no ('N'). Secondly, it updates the `ls_incdt_tms` variable to be the 29th February 2016. This date coincides with the last date in the purchases transactions date table. This is the last date a customer could have reported fraud but did not. This is important as when the purchases data is processed to create the feature matrix, the only transactions that will be of interest are those that occur before the date on which the fraud was reported. So in the case where no fraud is reported, the entire purchase period will be taken into account.

The final result is a table that consists of unique customer identifier, an indicator for fraud and a date which, if the fraud flag is positive, shows the date of the fraud incident.

4.4 Partitioning the Data

Normally, when the data is read into an EMR cluster using Spark, it is stored in an RDD - a resilient distributed dataset. It is through the RDD that data is split across the nodes of the EMR cluster. This can happen through either hash partitioning or range partitioning but will generally happen in a way to balance the data load across the cluster.

However, when Featuretools is used to combine the three tables into one big feature matrix, the data must sit in a Pandas DataFrame object and not in an RDD as this is a requirement of the Featuretools package. Another dependency of the package is that data must sit in memory to be processed (the curse of modularity), On the other hand, to leverage Spark's fast speed, the data must be in a format that enables it to be processed in parallel. To facilitate the parallel processing, the in-memory store and the dependency on the Pandas DataFrame, it is necessary to replicate the distributed storage of data and then use Spark's parallel processing command to facilitate the reading in and parallel processing of the functions. This step is implemented in a similar way to an RDD but using Pandas DataFrames and the S3 bucket.

MD5 hash encryption is a one-directional cryptofunction. It inputs a string character of any length and outputs a fixed-length value. Although its original use was to authenticate internet message, it has since been shown that it is not always secure and has given way to more secure methods of encryption. Given the same input (e.g. a `cst_id` number) it will always output the same value. Using the MD5 on `cst_id` - the only key that appears in all three tables

- means that it can divide out which customer information belongs in which partition.

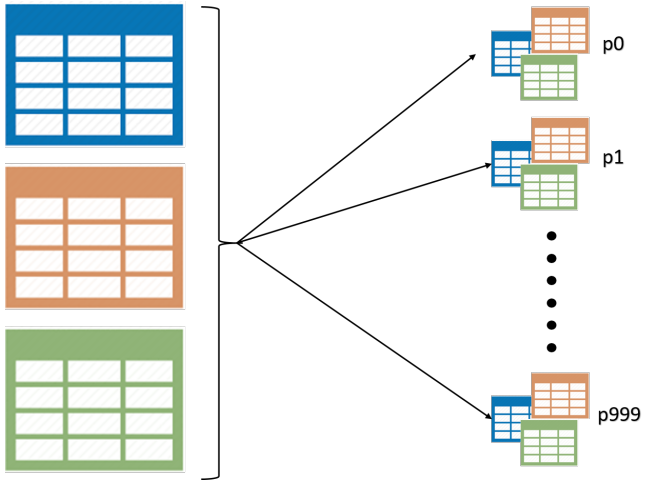


Figure 10: Diagram depicting Partitioning

The partitioning script (Appendix E) pulls the data in from the Redshift database and runs the data through these steps:

1. The input value of the MD5 algorithm must be a string, thus each of the tables `cst_id` variables are converted to a string and then converted to a value using the `id_to_hash` function.
2. A new column is created in each table called `part` which takes the hash values using a modulo divide by the total number of partitions (1000) to create a partition value within the range 0-999.
3. Next, a function (`write_files_to_partition`) acts on each of the dataframes. The function takes a dataframe as input and groups data rows by the newly created partition value. It then iterates through the partitioned values and writes each occurring subset to the correct partitioned file using the `s3fs` package. In addition, the function contains a timer which will display how long the code took to partition each table.

The result is one thousand file structures within the S3 bucket each containing a subset of the customer, fraud, and purchase transaction files.

4.5 Feature Synthesis on a Single Partition

To minimise cost and maximise efficiency, a single partition of data was used to explore and arrive at a final set of functions that could be applied to the larger distributed dataset. This was based on the assumption that the data is randomised and what will work for this partition, will work on all the partition.

The script for the section is contained in Appendix F and deals with the two main parts:

- Final cleaning and preprocessing steps on the data
- Using the Featuretools package to create a final dataframe of features

4.5.1 Final Cleaning

The first step needed is to clean and impute missing data from the customer and purchases tables.

The customer table using the `customer_preprocess` function is cleaned and imputed in the following way:

- Any missing values for the gross annual income number (`anual_grs_incm`) are replaced with the average income.
- Any missing values for a customers age (`age`) are replaced with the average age.
- Any missing values for a customers description (`hogan_cst_tp_desc`) are replaced with the most common class or mode.

The purchases table using the `purchases_clean` function is cleaned and imputed in the following way:

- Any missing values for the merchant city code (`mrcht_cty_code`) are replaced with a categorical value of `UKN` for 'unknown'
- Any missing values for the merchant category code (`cams_mrcht_cgy_tp_code`) are replaced with the most common class or mode.
- Any missing values for the merchant city name (`mrcht_cty_nm`) are replaced with a categorical value of `UKN` for 'unknown'.
- The following fields were kept in the table: Unique transaction id (`txn_id`), unique account identifier (`cc_ar_id`), unique customer identifier (`cst_id`), unique card identifier (`crd_id`), date of the transaction(`txn_date_con`), the merchant name (`mrcht_nm`), merchant city code (`mrcht_cty_code`), the merchant category code (`cams_mrcht_cgy_tp_code`) and the merchant city name (`mrcht_city_nm`). Any other fields were considered to be duplications, poorly populated or irrelevant.

4.5.2 Feature Synthesis

The last step is the feature synthesis using the open sourced package from Feature Labs: Featuretools. Much of the functionality of the improved deep

feature synthesis algorithm (DFS) was utilised. Time was spent refining and experimenting with ideas to get a 'good representation of the problem'. This replaces the industry step of 'acquiring domain knowledge' or sitting with a domain expert to understand what intuitive or already known data points may be useful or important. The discussion that follows gives an account of those primitives of Featuretools that were applied to the features finally chosen as the best representation of the data for the problem at hand. However, as previously mentioned, the focus of this dissertation is not finding the best features for the example problem, but on how to effectively obtain a feature matrix in the context of distributed big data.

Creating the Entity Set The initial step in feature synthesis is to set up an entity set and add the relevant tables to it. After cleaning and processing the data, the only two tables that need to be added are the customer table that contains demographic information and the purchases table. Note that the Fraud table has been reduced to only a fraud flag and the incident date-time features in the customer table. When adding tables to the entity set, a unique index for each must be specified and additionally, if applicable, a time index. In this case, the customer table has a unique key of the `cst_id` field while the purchase table has both a unique key `txn_id` and a time index which was the date the transaction was made (`txn_date_con`). (Figure 11) shows an excerpt from Appendix F which isolates the code for this step. Furthermore, the addition of the tables to the entity set specifies the data types for each field. This data type may be non-intuitive or different to the underlying data type. for instance, `crd_id` which was a numerical index for the card used on the transaction, becomes a categorical variable. Describing it this way allows the DFS algorithm to give more meaningful results e.g. the count of how many times a card identifier appears is more meaningful then summing them.

Define the relationships between tables The second step is to specify the relationship between the tables. This is done through a specification of primary and secondary keys between tables. For instance, indicating a primary key in one table through to a secondary key in another will infer a relationship between the tables. In the experimental data, the primary key of the customer data `cst_id` is specified as a secondary key in the purchases table (`cst_id`). This step does assume that there are keys in the tables that indicate relationships but as the package is designed to work from data generally stored in a relational database, this assumption is unlikely to be incorrect. Once the relationship is defined, it is added to the entity set.

Cut-off times Next, a cut-off time is gathered from the fraud table. This cut-off time is important as it represents a date before which, transactions are important and after which, irrelevant. i.e. if fraud was reported on a certain date, there is no value in looking at transactions that occurred after it. Adding this cut-off date to the DFS algorithm will allow it to only process data on the time index *before* this date.

```
In [239]: es.entity_from_dataframe(entity_id='customers', dataframe=customer,
                                   index = 'CST_ID',
                                   variable_types = {'AGE': vtypes.Categorical,
                                                     'HOGAN_CST_TP_DESC': vtypes.Categorical})

Out[239]: Entityset: fraud
            Entities:
              customers [Rows: 2019, Columns: 4]
            Relationships:
              No relationships

In [240]: es.entity_from_dataframe(entity_id='purchases', dataframe=purchases,
                                   index = 'TXN_ID',
                                   time_index = 'TXN_DATE_CON',
                                   variable_types = {"CC_AR_ID": vtypes.Categorical,
                                                     "CRD_ID": vtypes.Categorical,
                                       'MRCHT_NM':vtypes.Categorical,
                                       'MRCHT_CTY_CODE':vtypes.Categorical,
                                       'CAMS_MRCHT_CGY_TP_CODE':vtypes.Categorical,
                                       'MRCHT_CITY_NM':vtypes.Categorical })

Out[240]: Entityset: fraud
            Entities:
              customers [Rows: 2019, Columns: 4]
              purchases [Rows: 5193, Columns: 10]
            Relationships:
              No relationships

In [241]: r_purchases = ft.Relationship(es['customers']['CST_ID'], es['purchases']['CST_ID'])
          es.add_relationships([r_purchases])

Out[241]: Entityset: fraud
            Entities:
              customers [Rows: 2019, Columns: 4]
              purchases [Rows: 5193, Columns: 10]
            Relationships:
              purchases.CST_ID -> customers.CST_ID
```

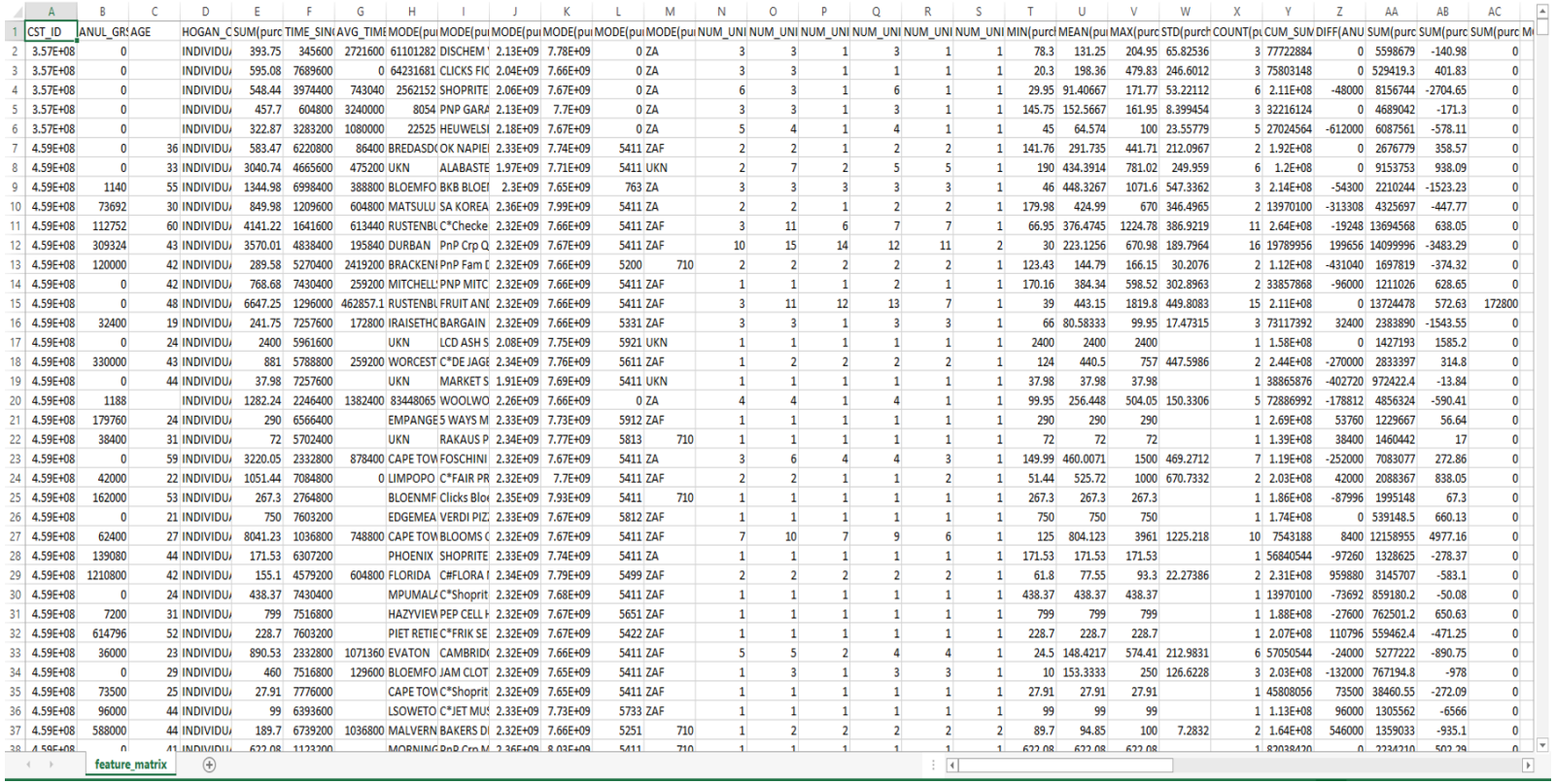
Figure 11: Code Extract

Primitives The DFS allows the user to specify the aggregations that they wish to perform within the DFS. In Featuretools language these are called "primitives" and are the functions such as sum, count that will be applied to the data to build the features. These "primitives" are split into two types: aggregation primitives versus transformation primitives. In simple terms, aggregation do just that: they create aggregations from the raw data such as counting the number of individual transactions a customer has made or giving a total of the transaction amounts. They can be thought of as the equivalent of "rolling up" of multiple data points into a single point. Transformative primitives on the other hand can be thought of as "breaking apart" data points - they take single variables and squeeze it for more information e.g. given the date of the transaction, it will add flags if the data was on a weekend, weekday etc. If the variable was numeric, such as the transaction value, it could flag if the amount was higher or lower than a certain noteworthy amount (e.g. was the amount more or less than the amount that the point of sale machine can authorise without dialling up to the network).

The only restriction on the primitives is the input format of the variable e.g. a "max" aggregation is not available to perform on a categorical variable. In addition, the output from the aggregations are standardised but Featuretools does allow for user designed or customer primitives. For a full list of primitives available, please see Appendix G.

Specifying the Target entity and maximum depth of DFS The last step is specifying the target entity for DFS i.e. to which level we would to like aggregate. For instance, if the business problem called for it, the demographic information could be brought down to the transaction level. However, in this case, the target entity is the customer table and the purchase transactions are aggregated up to the customer level. In more complex datasets, the maximum depth would need to be specified as well. i.e. how far down the relationship paths to travel.

Generating the feature matrix and a list of features The final step is brings all these items together and produce the final feature matrix and list of features. The results of this step are discussed in Chapter 5.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC
1	CST_ID	ANUL_GR	AGE	HOGAN_C	SUM(purc	TIME_SIN	AVG_TIM	MODE(pu	MODE(pu	MODE(pu	MODE(pu	MODE(pu	MODE(pu	NUM_UNI	NUM_UNI	NUM_UNI	NUM_UNI	NUM_UNI	NUM_UNI	MIN(purc	MEAN(pu	MAX(purc	STD(purch	COUNT(p	CUM_SUM	DIFF(ANU	SUM(purc	SUM(purc	SUM(purc	M
2	3.57E+08	0		INDIVIDU	393.75	345600	2721600	61101282	DISCHEM '	2.13E+09	7.78E+09	0	ZA	3	3	1	3	1	1	78.3	131.25	204.95	65.82536	3	77722884	0	5598679	-140.98	0	
3	3.57E+08	0		INDIVIDU	595.08	7689600	0	64231681	CLICKS FI	2.04E+09	7.66E+09	0	ZA	3	3	1	1	1	1	20.3	198.36	479.83	246.6012	3	75803148	0	529419.3	401.83	0	
4	3.57E+08	0		INDIVIDU	548.44	3974400	743040	2562152	SHOPRITE	2.06E+09	7.67E+09	0	ZA	6	3	1	6	1	1	29.95	91.40667	171.77	53.22112	6	2.11E+08	-48000	8156744	-2704.65	0	
5	3.57E+08	0		INDIVIDU	457.7	604800	3240000	8054	PNP GARA	2.13E+09	7.7E+09	0	ZA	3	3	1	3	1	1	145.75	152.5667	161.95	8.399454	3	32216124	0	4689042	-171.3	0	
6	3.57E+08	0		INDIVIDU	322.87	3283200	1080000	22525	HEUWELSI	2.18E+09	7.67E+09	0	ZA	5	4	1	4	1	1	45	64.574	100	23.55779	5	27024564	-612000	6087561	-578.11	0	
7	4.59E+08	0	36	INDIVIDU	583.47	6220800	86400	BREDASD(	OK NAPIE	2.33E+09	7.74E+09	5411	ZAF	2	2	1	2	2	1	141.76	291.735	441.71	212.0967	2	1.92E+08	0	2676779	358.57	0	
8	4.59E+08	0	33	INDIVIDU	3040.74	4665600	475200	UKN	ALABASTE	1.97E+09	7.71E+09	5411	UKN	2	7	2	5	5	1	190	434.3914	781.02	249.959	6	1.2E+08	0	9153753	938.09	0	
9	4.59E+08	1140	55	INDIVIDU	1344.98	6998400	388800	BLOEMFO	BKB BLOEI	2.3E+09	7.65E+09	763	ZA	3	3	3	3	3	1	46	448.3267	1071.6	547.3362	3	2.14E+08	-54300	2210244	-1523.23	0	
10	4.59E+08	73692	30	INDIVIDU	849.98	1209600	604800	MATSULU	SA KOREA	2.36E+09	7.99E+09	5411	ZA	2	2	1	2	2	1	179.98	424.99	670	346.4965	2	13970100	-313308	4325697	-447.77	0	
11	4.59E+08	112752	60	INDIVIDU	4141.22	1641600	613440	RUSTENBU	C*Checke	2.32E+09	7.66E+09	5411	ZAF	3	11	6	7	7	1	66.95	376.4745	1224.78	386.9219	11	2.64E+08	-19248	13694568	638.05	0	
12	4.59E+08	309324	43	INDIVIDU	3570.01	4838400	195840	DURBAN	PnP Crp Q	2.32E+09	7.67E+09	5411	ZAF	10	15	14	12	11	2	30	223.1256	670.98	189.7964	16	19789956	199656	14099996	-3483.29	0	
13	4.59E+08	120000	42	INDIVIDU	289.58	5270400	2419200	BRACKEN	PnP Fam [	2.32E+09	7.66E+09	5200	710	2	2	2	2	2	1	123.43	144.79	166.15	30.2076	2	1.12E+08	-431040	1697819	-374.32	0	
14	4.59E+08	0	42	INDIVIDU	768.68	7430400	259200	MITCHELL	PNP MITC	2.32E+09	7.66E+09	5411	ZAF	1	1	1	2	1	1	170.16	384.34	598.52	302.8963	2	33857868	-96000	1211026	628.65	0	
15	4.59E+08	0	48	INDIVIDU	6647.25	1296000	462857.1	RUSTENBU	FRUIT AN[	2.32E+09	7.66E+09	5411	ZAF	3	11	12	13	7	1	39	443.15	1819.8	449.8083	15	2.11E+08	0	13724478	572.63	172800	
16	4.59E+08	32400	19	INDIVIDU	241.75	7257600	172800	IRAISETH	BARGAIN	2.32E+09	7.66E+09	5331	ZAF	3	3	1	3	3	1	66	80.58333	99.95	17.47315	3	73117392	32400	2383890	-1543.55	0	
17	4.59E+08	0	24	INDIVIDU	2400	5961600		UKN	LCD ASH S	2.08E+09	7.75E+09	5921	UKN	1	1	1	1	1	1	2400	2400	2400		1	1.58E+08	0	1427193	1585.2	0	
18	4.59E+08	330000	43	INDIVIDU	881	5788800	259200	WORCEST	C*DE JAGI	2.34E+09	7.76E+09	5611	ZAF	1	2	2	2	2	1	124	440.5	757	447.5986	2	2.44E+08	-270000	2833397	314.8	0	
19	4.59E+08	0	44	INDIVIDU	37.98	7257600		UKN	MARKET S	1.91E+09	7.69E+09	5411	UKN	1	1	1	1	1	1	37.98	37.98	37.98		1	38865876	-402720	972422.4	-13.84	0	
20	4.59E+08	1188		INDIVIDU	1282.24	2246400	1382400	83448065	WOOLWO	2.26E+09	7.66E+09	0	ZA	4	4	1	4	1	1	99.95	256.448	504.05	150.3306	5	72886992	-178812	4856324	-590.41	0	
21	4.59E+08	179760	24	INDIVIDU	290	6566400		EMPANGE	5 WAYS M	2.33E+09	7.73E+09	5912	ZAF	1	1	1	1	1	1	290	290	290		1	2.69E+08	53760	1229667	56.64	0	
22	4.59E+08	38400	31	INDIVIDU	72	5702400		UKN	RAKAUS P	2.34E+09	7.77E+09	5813	710	1	1	1	1	1	1	72	72	72		1	1.39E+08	38400	1460442	17	0	
23	4.59E+08	0	59	INDIVIDU	3220.05	2332800	878400	CAPE TOV	FOSCHINI	2.32E+09	7.67E+09	5411	ZA	3	6	4	4	3	1	149.99	460.0071	1500	469.2712	7	1.19E+08	-252000	7083077	272.86	0	
24	4.59E+08	42000	22	INDIVIDU	1051.44	7084800	0	LIMPOPO	C*FAIR PR	2.32E+09	7.7E+09	5411	ZAF	2	2	1	1	2	1	51.44	525.72	1000	670.7332	2	2.03E+08	42000	2088367	838.05	0	
25	4.59E+08	162000	53	INDIVIDU	267.3	2764800		BLOENMF	Clicks Blo	2.35E+09	7.93E+09	5411	710	1	1	1	1	1	1	267.3	267.3	267.3		1	1.86E+08	-87996	1995148	67.3	0	
26	4.59E+08	0	21	INDIVIDU	750	7603200		EDGEMEA	VERDI PIZ	2.33E+09	7.67E+09	5812	ZAF	1	1	1	1	1	1	750	750	750		1	1.74E+08	0	539148.5	660.13	0	
27	4.59E+08	62400	27	INDIVIDU	8041.23	1036800	748800	CAPE TOV	BLOOMS (	2.32E+09	7.67E+09	5411	ZAF	7	10	7	9	6	1	125	804.123	3961	1225.218	10	7543188	8400	12158955	4977.16	0	
28	4.59E+08	139080	44	INDIVIDU	171.53	6307200		PHOENIX	SHOPRITE	2.33E+09	7.74E+09	5411	ZA	1	1	1	1	1	1	171.53	171.53	171.53		1	56840544	-97260	1328625	-278.37	0	
29	4.59E+08	1210800	42	INDIVIDU	155.1	4579200	604800	FLORIDA	C#FLORA I	2.34E+09	7.79E+09	5499	ZAF	2	2	2	2	2	1	61.8	77.55	93.3	22.27386	2	2.31E+08	959880	3145707	-583.1	0	
30	4.59E+08	0	24	INDIVIDU	438.37	7430400		MPUMALA	C*Shoprit	2.32E+09	7.68E+09	5411	ZAF	1	1	1	1	1	1	438.37	438.37	438.37		1	13970100	-73692	859180.2	-50.08	0	
31	4.59E+08	7200	31	INDIVIDU	799	7516800		HAZYVIEW	PEP CELL I	2.32E+09	7.67E+09	5651	ZAF	1	1	1	1	1	1	799	799	799		1	1.88E+08	-27600	762501.2	650.63	0	
32	4.59E+08	614796	52	INDIVIDU	228.7	7603200		PIET RETIE	C*FRIK SE	2.32E+09	7.67E+09	5422	ZAF	1	1	1	1	1	1	228.7	228.7	228.7		1	2.07E+08	110796	559462.4	-471.25	0	
33	4.59E+08	36000	23	INDIVIDU	890.53	2332800	1071360	EVATON	CAMBRID(	2.32E+09	7.66E+09	5411	ZAF	5	5	2	4	4	1	24.5	148.4217	574.41	212.9831	6	57050544	-24000	5277222	-890.75	0	
34	4.59E+08	0	29	INDIVIDU	460	7516800	129600	BLOEMFO	JAM CLOT	2.32E+09	7.65E+09	5411	ZAF	1	3	1	3	3	1	10	153.3333	250	126.6228	3	2.03E+08	-132000	767194.8	-978	0	
35	4.59E+08	73500	25	INDIVIDU	27.91	7776000		CAPE TOV	C*Shoprit	2.32E+09	7.65E+09	5411	ZAF	1	1	1	1	1	1	27.91	27.91	27.91		1	45808056	73500	38460.55	-272.09	0	
36	4.59E+08	96000	44	INDIVIDU	99	6393600		LSOWETO	C*JET MU	2.33E+09	7.73E+09	5733	ZAF	1	1	1	1	1	1	99	99	99		1	1.13E+08	96000	1305562	-6566	0	
37	4.59E+08	588000	44	INDIVIDU	189.7	6739200	1036800	MALVERN	BAKERS D	2.32E+09	7.66E+09	5251	710	1	2	2	2	2	2	89.7	94.85	100	7.2832	2	1.64E+08	546000	1359033	-935.1	0	
38	4.59E+08	0	41	INDIVIDU	622.08	1123200		MORNING	PnP Crp M	2.36E+09	8.03E+09	5411	710	1	1	1	1	1	1	622.08	622.08	622.08		1	82038420	0	2234210	502.29	0	

feature_matrix

Figure 12: Single Partition Feature Matrix

4.6 Scaling the Solution

Once a single partition of the data has been fully explored, the next step is to use these functions and knowledge on the AWS EMR cluster to scale the solution. For setup of the EMR cluster itself, please refer back to Section 4.2.4. Scaling the solution follows these steps:

1. Open up a Jupyter Lab notebook on the required cluster with a PySpark kernel.
2. Open a Spark Connect session.
3. Install required packages.
4. Write a function that loads the data from a single partition, run the cleaning and feature synthesis functions built for a single partition and push the final feature matrix back to the S3 bucket partition.
5. Test on a single partition.
6. Use the parallel processing and map functionality of Spark to iterate over all the partitions in S3.
7. Combine the partitioned feature matrix into a single table and push back to the S3 bucket as a single table.

The code used for this step can be found in Appendix H.

4.6.1 Set up the Jupyter Notebook with PySpark Kernel

To set up the Jupyter notebook a user needs to choose a cluster, security group and S3 bucket to which to write the log back. Connecting the notebook to the cluster requires that the cluster be in a "Waiting - Cluster ready" status. The user has an option to open straight into a Jupyter Notebook or open a Jupyter Lab environment. the Lab environment was used as it gives the option of multiple kernels (Python 3, PySpark, Spark or SparkR) as well as a console or terminal option.

4.6.2 Spark Context

Once the notebook environment is set up, a SparkSession can be opened. With the integration of PySpark and Spark on the EMR, there is no longer a need to specify a SparkContext as the interface to Spark. The simple SparkSession is a unified entry point that allows a user access to all of the spark functionality such as SparkContext, StreamingContext and SQLContext in a single step.

Figure 13: Starting a SparkSession

4.6.3 Installing Packages

There are several ways to install packages on the cluster:

- Install through the Bootstrap step on the setup of the AWS EMR.
- Install using the Jupyter Lab console option
- Install using the `sc.install_pypi_package` command

Of the three, the `sc.install_pypi_package` option was the most flexible and allowed additional packages to be installed as and when they were needed. However, it does require an installation for each new Notebook and SparkSession opened. After that, the user is required to import specific packages needed for their code using the normal `import` command.

4.6.4 Partition to Feature Matrix Function

The functions developed in the "Single Partition" section are used again here (i.e. the `customer_preprocess`, `purchases_clean` and `feature_engineering` functions) However, they are combined into a new function `feature_matrix` that reads in the three files from a single partition, applies all three functions and writes the resulting feature matrix to the same partition in the S3 bucket. The idea is to have a single function that can then be used to process all the partitions in a parallel manner.

To be able to read to and from the S3 buckets, the AWS user credentials (i.e. key and secret key) are needed as well as the s3fs package. The script is set up to run off a base directory specified outside of the function.

```
[4]: sc.list_packages()
```

```
Package                    Version
-------------------------- -------
beautifulsoup4             4.8.0
boto                       2.49.0
boto3                      1.11.9
botocore                   1.14.9
Click                      7.0
cloudpickle                1.2.2
dask                       2.10.1
DateTime                   4.3
distributed                2.10.0
docutils                   0.15.2
featuretools               0.13.1
fsspec                     0.6.2
HeapDict                   1.0.1
jmespath                   0.9.4
lxml                       4.4.1
msgpack                    0.6.2
mysqlclient                1.4.4
nltk                       3.4.5
nose                       1.3.4
numpy                      1.14.5
pandas                     1.0.0
pip                        20.0.2
psutil                     5.6.7
py-dateutil                2.2
python-dateutil            2.8.1
python36-sagemaker-pyspark 1.2.4
pytz                       2019.2
PyYAML                     5.3
s3fs                       0.4.0
s3transfer                 0.3.2
scipy                      1.4.1
setuptools                 45.1.0
six                        1.12.0
sortedcontainers           2.1.0
soupsieve                  1.9.3
tblib                      1.6.0
toolz                      0.10.0
tornado                    6.0.3
tqdm                       4.42.0
urllib3                    1.25.8
wheel                      0.34.1
windmill                   1.6
zict                       1.0.0
zope.interface             4.7.1
```

Figure 14: Checking installed packages using `sc.list_package`

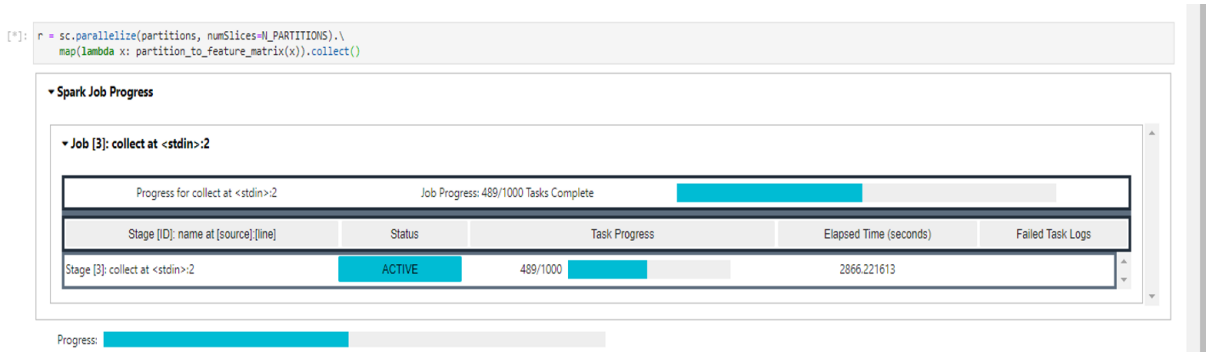


Figure 15: Figure showing the Spark Job Progress

4.6.5 Test for a Single Partition

Before setting up the parallel processing, the script tests the `feature_matrix` function on one partition. The first test was chosen at random but this step in the script became a handy tool to figure out problem partitions when the the parallel `feature_matrix` function failed.

4.6.6 Processes all the Partitions

The penultimate step is to process all the partitions in the S3 buckets Featuretools is reliant on the entity sets being Pandas DataFrames and thus the data cannot be read in as RDD's and then processed. In order to facilitate this, a combination of the `sc.parallel()` and `map` functions are used.

The `sc.parallel()` function works to read the data into a distributed dataset and create parallelized collections that can then be processed at the same time as each other. The map function will return a new distributed dataset by passing each of the elements through the feature matrix function. In addition, the action command `collect` was used to force the lazy execution.

When using the `sc.parallel()`, a user can either rely on Spark to set the number of partitions that the data is split into or can input their own. Here, the latter was chosen and supports the one thousand partitions that are sitting on the S3 buckets.

The job can then be monitored in the Jupyter Notebook by clicking on the Spark Job Progress. If the Spark job executes correctly, there should be a Feature Matrix in each S3 bucket partition.

4.6.7 Combine the Feature Matrices

And finally, to produce a consolidated dataset with all the features and another set with the labels for a supervised learning problem, the script combines all

the feature matrix dataframes into a single file written to the S3 bucket. In addition, it does the same to the fraud dataset where the 'Y/N' label for fraud sits. These two datasets are kept separate as normally the first step in a machine learning problem in a Python environment is to separate the X and Y values (although they are easily combinable due to their common `cst_id` key.)

This step is done via a function which reads in each dataset, combines the dataframes and writes out to the S3 bucket.

5 Results

This chapter discusses the results of the experiment and concentrates on three main aspects: how viable the infrastructure was, how long the experiment took to run and the effectiveness of the final feature matrix.

5.1 Infrastructure

On the whole, the AWS environment was easy to use and had a good documentation and support system. The infrastructure worked well, however several issues had to be solved throughout the experiment. Most of these were well documented and solutions were found either through the official documentation or StackOverflow issues posted by users with similar problems.

The AWS CLI load The first problem was that the AWS credentials that are needed must be exact - a single character incorrect will cause errors. In addition, copy and pasting from the credentials file to the console was not available. However, once the user key and secret access key are passed correctly the set-up works well. Having the AWS CLI installed also meant that when is was needed again e.g. to manage the s3fs package connection to the S3 bucket, the configuration file was already in use. Another point to note: if a user's credentials are reset, the configuration file must be reset as well.

The second problem was the raw data load from the AWS CLI. The 1.2 GB of purchase data took over 2 hours to complete. However, the problem didn't seem to be the tool but rather the slow internet speed of the WiFi network used. Once the ISP company had rectified the problem, the upload became much faster. However, the ISP package only allowed a 2 MB upload. The assumption is that a corporate would have a much faster connection speed and this would not be an issue.

The Redshift COPY statement can be problematic This step took the most research to get right. As the Redshift COPY command allows for multiple variations, the requirement was to find the *right* statement that worked in particular for the file being uploaded. For instance, the `CSV` command on the COPY statement won't work with either the `REMOVEQUOTE` or the `ESCAPE` command. This proved problematic in the purchases file where large amounts of non-standard text data was used. Eventually, it was imported not as a csv but as COPY with additional commands to make it work.

The EC2 instance was too small At first, a small instance of an EC2 instance was spun up and used - especially during the initial tests of the partitioning script. However, once the entire dataset was needed, the memory on such a small EC2 instance became a problem. This manifested itself through the purchases transaction dataset. The dataset was unable to load through the

psycopg2 connection and the EC2 instance was unable to process the write back to the partitions. Fortunately, the AWS infrastructure is extremely flexible and allows for the instance to be stopped, scaled and started again. See Figure 16.

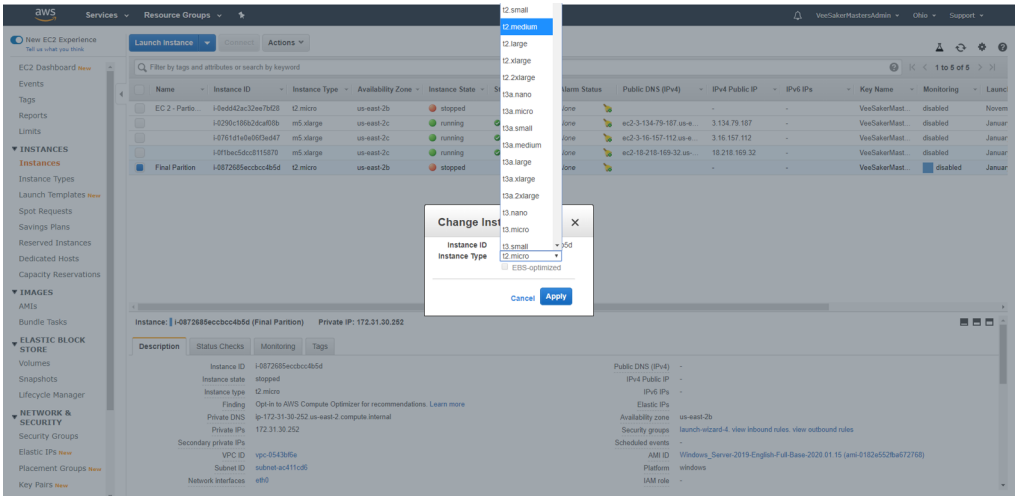


Figure 16: Scaling the EC2 Instance

Eventually, a t2.xlarge instance was used with 4CPU's and 16GB of memory available compared to the initial t2.micro with only 1CPU and 1GB memory. However, this instance is markedly more expensive than the t2.micro - $0.0196 per Hour vs. $0.24 per Hour. i.e. For 8 hours of use, the t2.micro will cost less than 15 US cents versus almost $2 for the t2.xlarge. A word of warning: keep a close eye on the billing dashboard as leaving an EC2 instance running will quickly ramp up costs. There is no need to delete the instance but it is a good idea to pause the instance through the "stop" action when one is not actively using it.

The SparkSession kept timing out The next memory problem occurred during the parallelization step on the Spark run. The Spark job progress would get to approximately job 678 and then stop due to SparkSession time-out error. In other words, rather then it being an issue with the code, the connection to Spark would close and stop the job. Research into this issue eventually pointed to the driver running out of memory. Once again, switching the underlying instances solved the problem. It was slightly harder to do this on an EMR than on the EC2. The user can clone the current EMR setup with steps and then opt for bigger Master and Slave nodes. Once the cluster has spun up, it needs to stop and reallocate the notebook environment to the new cluster. The notebook keeps all the saved scripts and Jupyter Lab elements so apart from about 5 minutes while the new cluster starts, the process is fairly seamless. Once again, as soon as one is not using the EMR cluster, one should opt to terminate it to keep costs from escalating.

```
In [16]: write_files_to_partition(fraud, "fraud", progress = 10)

         99.0% complete. 216 seconds elapsed.onds elapsed.
         2022971 rows processed in 218 seconds.

In [17]: write_files_to_partition(customer, "customer", progress = 10)

         99.0% complete. 203 seconds elapsed.onds elapsed.
         2022971 rows processed in 204 seconds.

In [18]: write_files_to_partition(purchases, "purchases", progress = 10)

         99.0% complete. 681 seconds elapsed.onds elapsed..
         5313831 rows processed in 687 seconds.
```

Figure 17: Run times of Partitioning Files to S3

5.2 Time Efficiency

One of the most important elements in the experiment was the time that it took to run. The initial premise was to save time in the analytics process. The following section looks at each point in the infrastructure and the run time. However, it does not include the time it took to research and decide on each element. The results only consider the time for an end to end run of the solution as once the infrastructure is set up, it is available for other problems and data. In addition, as previously stated, it is assumed that data would already be in the Reshift database at the start of a problem and so the time taken to upload data to Redshift is not included. Likewise, the step which looks at the feature synthesis and cleaning on a single partition is not included. The timing of that step would largely depend on the problem to be solved, the experience of the analyst and knowledge of the domain or access to a domain expert and would be entirely variable. In addition, that step would take place in any version of a machine learning problem.

The time for the framework to run, broken down by task, is shown in Table 5. The total time to run is approximately 200 minutes or approximately 3 hours and twenty minutes.

5.3 Results of the Feature Matrix

This section examines the Feature Matrix itself and the features that were produced. In total, 75 features were generated (see Table 10). Although not an exhaustive list of what Featuretools can produce, it has generated these 75 features from the 12 raw features that were fed into the DFS algorithm. Given that the overall process ran for around 3.5 hours, and that there were already 3 features on the customer level, this is about 2.8 minutes to generate a feature.

Table 5: Framework Run Time

Task	Subtask	Environment	Time Taken
Initial Cleaning Script		Redshift Database	2.43 mins
Write Data to S3 bucket	Purchase Data	Redshift Database	10.36 mins
	Fraud Data	Redshift Database	3.68 mins
	Customer Data	Redshift Database	3.45 mins
Parallelized Feature Matrix	Single Partition	EMR\Spark	6.32 secs (*)
	Entire Dataset	EMR\Spark	97.68 mins
Joining Partitioned Tables	Per partition	EMR\Spark	4.21 secs (*)
	Entire Dataset	EMR\Spark	79.82 mins
Total Time (*excluding single partitions)			**197.42 mins**

The names of the features are generated from the DFS algorithm as well and it is probably worth the time to rename these to something more descriptive and intuitive to the analyst or business stakeholder. Figure 12 shows what the final feature matrix looks like for a single partition.

To show the accuracy of the feature matrix, multiple random customers were selected and the accuracy of the computed features checked. Below is presented an example of the validation that was conducted using `cst_id` of 461234003 (henceforth known as Bob). Bob's data from the three individual tables is shown in Tables 6, 7 and 8. Bob is an individual customer aged 39, with a gross annual income of R160,632 a year. He has reported no fraud in the period and has made only one purchase of R33.98 at Game in Fourways on the 5th December.

In comparison, Table 9 shows a view of the features generated for Bob. These were easily verifiable. As far as accuracy goes, Bob's data is complete and accurate.

For instance, `Feature: NUM_UNIQUE(purchases.CRD_ID)` which reflects a 1. As Bob only made one transaction, there should only be one card that was used. The same goes for `Feature:SUM(purchases.TXN_AMT)` which reflects the R33.98 that was spent by Bob. However, other features seem less informative or useful. For instance, `Feature:CUM_SUM(ANUL_GRS_INCM)` which has calculated the cumulative sum of all customer's income. This may be helpful in comparison to Bob's income but not in its current form. This may be a result of the analyst's lack of knowledge on FeatureTools but could be useful if another calculation was added (e.g. the ratio of Bob's income to the total).

However, the solution has managed to generate many features. If the aim to synthesize as many features as a pre-cursor to feature reduction or selection, then this solution is ideal. If the aim is to produce features that can aide interpretation of a model, then the analyst must spend time refining the primitives used.

Table 6: Sample customer data: Bob's data

Variable Name	**Value**
cst_id	461234003
age	39
anul_grs_incm	160632
hogan_cst_tp_desc	INDIVIDUAL

Table 7: Sample fraud data: Bob's data

cst_id	**frd_flg**	**cut_off_time**
461234003	N	29/02/2016

Table 8: Sample purchase transaction data: Bob's data

Variable Name	**Value**
tm_prd_id	148264
cc_ar_id	7678210324
cst_id	461234003
crd_id	2324986171
txn_no	2.29074E+29
txn_date_con	42343
date_rcvd	42345
txn_amt_orig_ccy	33.98
txn_amt	33.98
mrcht_no	922260
mrcht_nm	GAME FOURWAYS 126
mrcht_cty_code	ZAF
cams_mrcht_cgy_tp_code	5311
cams_mrcht_cgy_tp_desc	DEPARTMENT STOR
mrcht_str_adr	
mrcht_city_nm	FOURWAYS

Table 9: Feature Matrix Values for Bob

Feature Number	Feature Name	Value
0	<Feature: ANUL_GRS_INCM>	160632
1	<Feature: AGE>	39
2	<Feature: HOGAN_CST_TP_DESC>	INDIVIDUAL
3	<Feature: SUM(purchases.TXN_AMT)>	33.98
4	<Feature: TIME_SINCE_LAST(purchases.TXN_DATE_CON)>	7430400
5	<Feature: AVG_TIME_BETWEEN(purchases.TXN_DATE_CON)>	
6	<Feature: MODE(purchases.MRCHT_CITY_NM)>	FOURWAYS
7	<Feature: MODE(purchases.MRCHT_NM)>	GAME FOURWAYS 126
8	<Feature: MODE(purchases.CRD_ID)>	2324986171
9	<Feature: MODE(purchases.CC_AR_ID)>	7678210324
10	<Feature: MODE(purchases.CAMS_MRCHT_CGY_TP_CODE)>	5311
11	<Feature: MODE(purchases.MRCHT_CTY_CODE)>	ZAF
12	<Feature: NUM_UNIQUE(purchases.MRCHT_CITY_NM)>	1
13	<Feature: NUM_UNIQUE(purchases.MRCHT_NM)>	1
14	<Feature: NUM_UNIQUE(purchases.CRD_ID)>	1
15	<Feature: NUM_UNIQUE(purchases.CC_AR_ID)>	1
16	<Feature: NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE)>	1
17	<Feature: NUM_UNIQUE(purchases.MRCHT_CTY_CODE)>	1
18	<Feature: MIN(purchases.TXN_AMT)>	33.98
19	<Feature: MEAN(purchases.TXN_AMT)>	33.98
20	<Feature: MAX(purchases.TXN_AMT)>	33.98
21	<Feature: STD(purchases.TXN_AMT)>	
22	<Feature: COUNT(purchases)>	1
23	<Feature: CUM_SUM(ANUL_GRS_INCM)>	155192376
24	<Feature: DIFF(ANUL_GRS_INCM)>	112632
25	<Feature: SUM(purchases.CUM_SUM(TXN_AMT))>	862033.9
26	<Feature: SUM(purchases.DIFF(TXN_AMT))>	15
27	<Feature: SUM(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))>	0
28	<Feature: MODE(purchases.WEEKDAY(TXN_DATE_CON))>	5
29	<Feature: MODE(purchases.DAY(TXN_DATE_CON))>	5
30	<Feature: MODE(purchases.MONTH(TXN_DATE_CON))>	12
31	<Feature: NUM_UNIQUE(purchases.WEEKDAY(TXN_DATE_CON))>	1
32	<Feature: NUM_UNIQUE(purchases.DAY(TXN_DATE_CON))>	1
33	<Feature: NUM_UNIQUE(purchases.MONTH(TXN_DATE_CON))>	1
34	<Feature: MIN(purchases.CUM_SUM(TXN_AMT))>	862033.9
35	<Feature: MIN(purchases.DIFF(TXN_AMT))>	15
36	<Feature: MIN(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))>	0
37	<Feature: MEAN(purchases.CUM_SUM(TXN_AMT))>	862033.9
38	<Feature: MEAN(purchases.DIFF(TXN_AMT))>	15
39	<Feature: MEAN(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))>	0
40	<Feature: MAX(purchases.CUM_SUM(TXN_AMT))>	862033.9
41	<Feature: MAX(purchases.DIFF(TXN_AMT))>	15
42	<Feature: MAX(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))>	0
43	<Feature: STD(purchases.CUM_SUM(TXN_AMT))>	
44	<Feature: STD(purchases.DIFF(TXN_AMT))>	
45	<Feature: STD(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))>	
46	<Feature: CUM_SUM(NUM_UNIQUE(purchases.CRD_ID))>	2093
47	<Feature: CUM_SUM(MEAN(purchases.TXN_AMT))>	516195.3148
48	<Feature: CUM_SUM(NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE))>	2300
49	<Feature: CUM_SUM(NUM_UNIQUE(purchases.MRCHT_CTY_CODE))>	1309
50	<Feature: CUM_SUM(SUM(purchases.TXN_AMT))>	1328529.89
51	<Feature: CUM_SUM(MIN(purchases.TXN_AMT))>	327777.63
52	<Feature: CUM_SUM(STD(purchases.TXN_AMT))>	
53	<Feature: CUM_SUM(NUM_UNIQUE(purchases.MRCHT_CITY_NM))>	2193
54	<Feature: CUM_SUM(COUNT(purchases))>	2979
55	<Feature: CUM_SUM(NUM_UNIQUE(purchases.CC_AR_ID))>	2560
56	<Feature: CUM_SUM(AVG_TIME_BETWEEN(purchases.TXN_DATE_CON))>	
57	<Feature: CUM_SUM(MAX(purchases.TXN_AMT))>	884351.04
58	<Feature: CUM_SUM(TIME_SINCE_LAST(purchases.TXN_DATE_CON))>	5128704000
59	<Feature: CUM_SUM(NUM_UNIQUE(purchases.MRCHT_NM))>	2738
60	<Feature: DIFF(NUM_UNIQUE(purchases.CRD_ID))>	-2
61	<Feature: DIFF(MEAN(purchases.TXN_AMT))>	-299.989
62	<Feature: DIFF(NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE))>	-4
63	<Feature: DIFF(NUM_UNIQUE(purchases.MRCHT_CTY_CODE))>	0
64	<Feature: DIFF(SUM(purchases.TXN_AMT))>	-3305.71
65	<Feature: DIFF(MIN(purchases.TXN_AMT))>	-90.92
66	<Feature: DIFF(STD(purchases.TXN_AMT))>	
67	<Feature: DIFF(CUM_SUM(ANUL_GRS_INCM))>	160632
68	<Feature: DIFF(NUM_UNIQUE(purchases.MRCHT_CITY_NM))>	-3
69	<Feature: DIFF(COUNT(purchases))>	-9
70	<Feature: DIFF(NUM_UNIQUE(purchases.CC_AR_ID))>	-6
71	<Feature: DIFF(AVG_TIME_BETWEEN(purchases.TXN_DATE_CON))>	
72	<Feature: DIFF(MAX(purchases.TXN_AMT))>	-797.48
73	<Feature: DIFF(TIME_SINCE_LAST(purchases.TXN_DATE_CON))>	4147200
74	<Feature: DIFF(NUM_UNIQUE(purchases.MRCHT_NM))>	-9

Table 10: Final Features Generated

No.	Feature Name	Description	Level
0	ANUL_GRS_INCM	Annual Gross Income	Observation
1	AGE	Age	Observation
2	HOGAN_CST_TP_DESC	Type of Customer	Observation
3	SUM(purchases.TXN_AMT)	Total Transaction Amount	Observation
4	TIME_SINCE_LAST(purchases.TXN_DATE_CON)	Time since Last Transaction	Observation
5	AVG_TIME_BETWEEN(purchases.TXN_DATE_CON)	Average Time between Transactions	Observation
6	MODE(purchases.MRCHT_CITY_NM)	Most Common Merchant Name	Observation
7	MODE(purchases.MRCHT_NM)	Most Common Merchant Number	Observation
8	MODE(purchases.CRD_ID)	Most Common Card Used	Observation
9	MODE(purchases.CC_AR_ID)	Most Common Account Used	Observation
10	MODE(purchases.CAMS_MRCHT_CGY_TP_CODE)	Most Common Type of Merchant	Observation
11	MODE(purchases.MRCHT_CTY_CODE)	Most Common City Code	Observation
12	NUM_UNIQUE(purchases.MRCHT_CITY_NM)	Number of Unique Cities Names	Observation
13	NUM_UNIQUE(purchases.MRCHT_NM)	Number of Unique Merchants Names	Observation
14	NUM_UNIQUE(purchases.CRD_ID)	Number of Unique Cards used	Observation
15	NUM_UNIQUE(purchases.CC_AR_ID)	Number of Accounts	Observation
16	NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE)	Number of Unique Merchant Types	Observation
17	NUM_UNIQUE(purchases.MRCHT_CTY_CODE)	Number of Unique Cities Code	Observation
18	MIN(purchases.TXN_AMT)	Lowest Transaction Amount	Observation
19	MEAN(purchases.TXN_AMT)	Average Transaction Amount	Observation
20	MAX(purchases.TXN_AMT)	Highest Transaction Amount	Observation
21	STD(purchases.TXN_AMT)	Std. Deviation of Transaction	Observation
22	COUNT(purchases)	Number of Transactions	Observation
23	CUM_SUM(ANUL_GRS_INCM)	Cumulative Sum of Income	Population
24	DIFF(ANUL_GRS_INCM)	Difference in Income	Difference to Previous Observation
25	SUM(purchases.CUM_SUM(TXN_AMT))	Cumulative Sum of Transaction Amount	Population
26	SUM(purchases.DIFF(TXN_AMT))	Difference in Transaction Amount	Difference to Previous Observation
27	SUM(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))	Time since Previous Transaction	Observation
28	MODE(purchases.WEEKDAY(TXN_DATE_CON))	Most Common Weekday of Transaction	Observation
29	MODE(purchases.DAY(TXN_DATE_CON))	Most Common Day of Transaction	Observation
30	MODE(purchases.MONTH(TXN_DATE_CON))	Most Common Month of Transaction	Observation
31	NUM_UNIQUE(purchases.WEEKDAY(TXN_DATE_CON))	Number of Unique Transactions in a week	Observation
32	NUM_UNIQUE(purchases.DAY(TXN_DATE_CON))	Number of Unique Transactions in Day	Observation
33	NUM_UNIQUE(purchases.MONTH(TXN_DATE_CON))	Number of Unique Transactions in a Month	Observation
34	MIN(purchases.CUM_SUM(TXN_AMT))	Min of Cumulative Sum of Transactions	Population
35	MIN(purchases.DIFF(TXN_AMT))	Min of Difference in Transaction Amount	Difference to Previous Observation
36	MIN(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))	Shortest Time between Transaction	Observation
37	MEAN(purchases.CUM_SUM(TXN_AMT))	Average Cumulative Sum of Transaction Amount	Population
38	MEAN(purchases.DIFF(TXN_AMT))	Average Difference in Transaction Amount	Difference to Previous Observation
39	MEAN(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))	Average Time between Transaction	Observation
40	MAX(purchases.CUM_SUM(TXN_AMT))	Max Difference in Transaction Amounts	Population
41	MAX(purchases.DIFF(TXN_AMT))	Max Difference in Transaction Amounts	Difference to Previous Observation
42	MAX(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))	Maximum Time between Transaction	Observation
43	STD(purchases.CUM_SUM(TXN_AMT))	Std. Deviation of Cumulative Sum of Transaction Amount	Population
44	STD(purchases.DIFF(TXN_AMT))	Std Deviation of Transaction Amount difference	Difference to Previous Observation
45	STD(purchases.TIME_SINCE_PREVIOUS(TXN_DATE_CON))	Std Deviation of Time since Previous Transaction	Observation
46	CUM_SUM(NUM_UNIQUE(purchases.CRD_ID))	Cumulative Sum of Number of Cards	Population

47	CUM_SUM(MEAN(purchases.TXN_AMT))	Cumulative Sum of Average Transaction value	Population
48	CUM_SUM(NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE))	Cumulative Sum Unique Transactions of Merchant Code	Population
49	CUM_SUM(NUM_UNIQUE(purchases.MRCHT_CTY_CODE))	Cumulative Sum of Unique Transactions of Merchant City	Population
50	CUM_SUM(SUM(purchases.TXN_AMT))	Cumulative Sum of the Sum of Transaction Amount	Population
51	CUM_SUM(MIN(purchases.TXN_AMT))	Cumulative Sum of the Min Transaction Amounts	Population
52	CUM_SUM(STD(purchases.TXN_AMT))	Cumulative Sum of the Std Deviation of Transaction Amounts	Population
53	CUM_SUM(NUM_UNIQUE(purchases.MRCHT_CITY_NM))	Cumulative Sum of Number of Unique Transaction	Population
54	CUM_SUM(COUNT(purchases))	Total number of Purchases	Population
55	CUM_SUM(NUM_UNIQUE(purchases.CC_AR_ID))	Cumulative Number of Cards used	Population
56	CUM_SUM(AVG_TIME_BETWEEN(purchases.TXN_DATE_CON))	Cumulative Sum of Average Time between Transactions	Population
57	CUM_SUM(MAX(purchases.TXN_AMT))	Cumulative Sum of Transaction Amount	Population
58	CUM_SUM(TIME_SINCE_LAST(purchases.TXN_DATE_CON))	Cumulative Sum of Time since Last Purchase	Population
59	CUM_SUM(NUM_UNIQUE(purchases.MRCHT_NM))	Cumulative Sum of Number of Unique Merchant Name	Population
60	DIFF(NUM_UNIQUE(purchases.CRD_ID))	Difference in Number of Cards Used	Difference to Previous Observation
61	DIFF(MEAN(purchases.TXN_AMT))	Average Difference in Transaction Amount	Difference to Previous Observation
62	DIFF(NUM_UNIQUE(purchases.CAMS_MRCHT_CGY_TP_CODE))	Difference in Number of Unique Merchants Types	Difference to Previous Observation
63	DIFF(NUM_UNIQUE(purchases.MRCHT_CTY_CODE))	Difference in Number of Unique Cities Codes transacted in	Difference to Previous Observation
64	DIFF(SUM(purchases.TXN_AMT))	Difference in Total Transaction Amounts	Difference to Previous Observation
65	DIFF(MIN(purchases.TXN_AMT))	Difference in Minimum Transaction amounts	Difference to Previous Observation
66	DIFF(STD(purchases.TXN_AMT))	Difference in Std Deviations of Transaction Amounts	Difference to Previous Observation
67	DIFF(CUM_SUM(ANUL_GRS_INCM))	Difference in Annual Gross Income	Difference to Previous Observation
68	DIFF(NUM_UNIQUE(purchases.MRCHT_CITY_NM))	Difference in Number of Unique City Names	Difference to Previous Observation
69	DIFF(COUNT(purchases))	Difference in Number of purchases	Difference to Previous Observation
70	DIFF(NUM_UNIQUE(purchases.CC_AR_ID))	Difference in Number of Accounts used	Difference to Previous Observation
71	DIFF(AVG_TIME_BETWEEN(purchases.TXN_DATE_CON))	Difference in Average Time between Transactions	Difference to Previous Observation
72	DIFF(MAX(purchases.TXN_AMT))	Difference in Maximum Transaction Amounts	Difference to Previous Observation
73	DIFF(TIME_SINCE_LAST(purchases.TXN_DATE_CON))	Difference in Time since Last Purchased	Difference to Previous Observation
74	DIFF(NUM_UNIQUE(purchases.MRCHT_NM))	Difference in Number of Unique Merchant Names	Difference to Previous Observation

6 Conclusion

6.1 Summary

This dissertation investigated the initial stages of data analytics, and in particular the automated generation of new features from raw data. The first task is to structure a set of good quality data that best represents the real-world problem - generally called feature engineering. A single file is ultimately required for machine learning which must hold information about complex relationships across multiple relations that are often of differing granularity.

The feature engineering task is often under-appreciated and time-consuming but comes with many benefits. It is considered crucial in the delivery of a good prediction or model and generally involves feature synthesis followed by feature reduction and selection.

Featuretools is an open-source package that uses the Deep Feature Synthesis (DFS) algorithm to combine tables across granularity and relationships into a single flat form table.

Unfortunately, it comes with two major limitations: it requires data to be stored in memory to process and is heavily reliant on the Pandas DataFrame to work. These limitations are major restrictions when used with tools such as Apache Spark which stores data across a cluster of machines.

This dissertation investigated the viability and effectiveness of automated feature synthesis using cloud-computing resources. A framework for achieving this was proposed and tested using a real-world fraud prediction example. AWS infrastructure was set up and data loaded into Redshift - an AWS distributed relational database. The data consisted of three separate tables representing three different entities - customers, fraud cases and purchase transactions. The customer and fraud tables were at a customer level while the purchase transaction table was at a lower granular level with multiple records of the same customer occurring. Initial cleaning took place and then data was passed into an AWS EC2 instance. Target labels were generated for the entire customer set from the fraud table and a key that was common across all three tables was hashed to a partition number, to ensure that all data for a customer resided on the same partition. These were written to individual folders within the S3 bucket with the help of the popular s3fs Python package. As prescribed by the framework, a single partition was explored first, and then AWS Elastic MapReduce used to obtain the final feature matrix using parallel processing. The feature matrix produced 75 features from 12 input variables and was time efficient with a total end-to-end run time of 3.5 hours and a cost of approximately R 814.

6.2 Lessons Learnt

There are several insights gained in the course of this work with regard to the viability of the infrastructure and the effectiveness of the automated feature synthesis tool itself.

6.2.1 Infrastructure

Firstly, the infrastructure is not as easy to set up as many "how-to guides" imply. The more components used and the more times the data has to be passed between components, the more complex the data pipeline becomes. Many iterations were required to correct problems encountered.For instance, AWS credentials must be passed correctly, the right ports open for connections and the right sized instances or environments set up.

On the positive side, AWS is a very flexible platform. On two occasions, when the memory of the component was not sufficient to cope with the size of the data, it was a fairly painless process to rescale it. On both occasions, there was minimal downtime on the pipeline and no work was lost during the changeover.

Lastly, while the experiment showed that the automated framework was not too costly for data of that size, cost does need to be kept in mind. It is important to keep an eye on the billing dashboard and to make sure any unused instances of either the AWS EMR or EC2 are stopped or terminated to avoid fees.

6.2.2 Common Identifier

In this example, there was a common identifier across all three tables - `cst_id`. It existed either as a primary or secondary key in each table and so made an ideal variable on which to run the MD5. However, if a dataset does not have a common identifier, one would have to be created across the relations by some method. There must be a common key feature in all input relations in order to partition data in such a way as to overcome the curse of dimensionality.

6.2.3 Feature Synthesis and Data Processing

The experiment did successfully produce a variety of synthesised features for a distributed set of data on the AWS cloud. It took three tables that were large in nature, transformed and fitted them for the business problem and produced a set of automatically generated features. In a practical application, the next step would be to assess the appropriateness of these features in terms of adding value and understanding how these impact the target variable. Feature selection or reduction techniques would be employed, and the process repeated until satisfactory accuracy of the overall machine learning pipeline was achieved. This dissertation did not set out to solve the business problem itself but to illustrate how feature generation could be accomplished. The feature matrix

generation was thus assessed based on the viability of its automation in terms of run time, rather than its suitability for solving the problem at hand

The tool proved viable to use in this context, and the experiment produced some valuable and meaningful features. In cases where the goal is to generate as many features as possible as input into a feature selection or feature reduction algorithm, more is clearly better than less. The feature selection can then provide more valuable and meaningful input to whichever machine learning approach is subsequently used.

The entire data pipeline runs for approximately 3.5 hours to completion for an initial 1.3 GB dataset (split between three relations) and cost R 814 (approximately $52). However, the development and setting up of the pipeline (i.e cumulatively getting each function and cloud computing component to work and communicate, etc.) took much longer and was more costly. However, for another problem and dataset, only two steps of the proposed framework would need to be redeveloped to accommodate the new dataset. The initial cleaning would need to be redeveloped specifically to the issues within the dataset. And the development of the feature matrix and preprocessing functions on a single partition would need to be realigned to the new problem. These redeveloped functions could then easily be pushed into the Spark job for scalability. On balance, this is a fairly effective pipeline in terms of analyst's time, when compared to how long it would take to manually code feature synthesis in Redshift.

It should nevertheless be noted that this does not mean that the process can completely replace the domain expertise that a truly effective model would take into account. This solution serves as a tool to implement and replace the coding needed to generate features, but there is still value in understanding the business domain and brainstorming possible valuable features and incorporating it into the process.

6.3 Future Work

This dissertation highlights some areas that can be explored further.

The first obvious extension would be to continue with the feature selection and machine learning steps that follow feature synthesis. It is suggested that full data exploration, feature selection, and (where necessary) data resampling should be undertaken before a machine learning algorithm is applied.

It would be beneficial to conduct a similar experiment using a different infrastructure or different cloud computing platform such as Azure.

In addition, more complex entity sets with a larger number of tables and relationships should be investigated, to determine how the solution scales for tables and relationships that had a depth of four or more.

6.4 Concluding Remarks

This dissertation has focused on the initial steps that an analyst must carry out in order to prepare a dataset for a supervised learning problem. It has taken a large dataset and effectively used an open-source package known for its ingenious feature generation ability and combined it with a cloud computing platform to overcome processing power issues. The framework proposed in this work was shown to be effective and viable for automated feature synthesis in cloud computing.

References

AWS. (2020). *AWS homepage.* Retrieved 07-02-2020, from https://aws.amazon.com/

Azure, M. (2020). *Microsoft Azure homepage.* Retrieved 07-02-2020, from https://azure.microsoft.com/en-us/

Bahnsen, A. C., Aouada, D., Stojanovic, A., & Ottersten, B. (2016). Feature engineering strategies for credit card fraud detection. *Expert Systems with Applications*, *51*, 134–142.

Bakshi, A. (2019). *Mapreduce example: Reduce side join in hadoop mapreduce.* Retrieved 07-02-2020, from https://www.edureka.co/blog/mapreduce-example-reduce-side-join/

Bellman, R., et al. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, *60*(6), 503–515.

Beyer, M. A., & Laney, D. (2012). The importance of 'big data': a definition. *Stamford, CT: Gartner*, 2014–2018.

Borthakur, D., et al. (2008). HDFS architecture guide. *Hadoop Apache Project*, *53*(1-13), 2.

Chambers, B., & Zaharia, M. (2018). *Spark: The definitive guide: Big data processing made simple.* " O'Reilly Media, Inc.".

Cheng, W., Kasneci, G., Graepel, T., Stern, D., & Herbrich, R. (2011). Automated feature generation from structured knowledge. In *Proceedings of the 20th acm international conference on information and knowledge management* (pp. 1395–1404).

Dataflair. (2019). *Hadoop Ecosystem and their components – a complete tutorial.* Retrieved 07-02-2020, from https://data-flair.training/blogs/hadoop-ecosystem-components/

Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Domingos, P. M. (2012). A few useful things to know about machine learning. *Commun. ACM*, *55*(10), 78–87.

dotdata. (2020). *dotdata homepage.* Retrieved 07-02-2020, from https://dotdata.com/

FeatureLabs. (2020). *Featurelabs.* Retrieved 07-02-2020, from https://www.featurelabs.com/

Featuretools. (2020). *Featuretools.* Retrieved 07-02-2020, from https://www.featuretools.com/

Fodor, I. K. (2002). *A survey of dimension reduction techniques* (Tech. Rep.). Lawrence Livermore National Lab., CA (US).

Foundation, A. S. (2019). *Apache Hadoop.* Retrieved 25-10-2019, from https://hadoop.apache.org/

García, S., Ramírez-Gallego, S., Luengo, J., Benítez, J. M., & Herrera, F. (2016). Big data preprocessing: methods and prospects. *Big Data Analytics*, *1*(1), 9.

Gary, C. (2019). *A deep dive into data quality.* Retrieved 07-02-2020, from https://towardsdatascience.com/a-deep-dive-into-data-quality-c1d1ee576046

GoogleCloud. (2020a). *Google cloud homepage.* Retrieved 07-02-2020, from https://cloud.google.com/

GoogleCloud. (2020b). *Khan academy: Scaling and simplifying.* Re-

trieved 07-02-2020, from https://cloud.google.com/customers/khan-academy//

Güven, A., Polat, K., Kara, S., & Güneş, S. (2008). The effect of generalized discriminate analysis (GDA) to the classification of optic nerve disease from vep signals. *Computers in Biology and medicine*, *38*(1), 62–68.

h20.ai. (2020). *h20.ai homepage.* Retrieved 07-02-2020, from https://www.h2o.ai/

Heaton, J. (2016). An empirical analysis of feature engineering for predictive modeling. In *Southeastcon 2016* (pp. 1–6).

Hugo, B.-A. (2018). *Machine learning with kaggle: Feature engineering.* Retrieved 07-02-2020, from https://www.datacamp.com/community/tutorials/feature-engineering-kaggle

Kanter, J. M., & Veeramachaneni, K. (2015). Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE international conference on data science and advanced analytics (DSAA)* (pp. 1–10).

Khurana, U., Turaga, D., Samulowitz, H., & Parthasrathy, S. (2016). Cognito: Automated feature engineering for supervised learning. In *2016 IEEE 16th international conference on data mining workshops (ICDMW)* (pp. 1304–1307).

Kimball, R., & Caserta, J. (2004). *The data warehouse ETL toolkit*. John Wiley & Sons.

Koehrsen, W. (2018). *Featuretools on Spark.* Retrieved 07-02-2020, from https://blog.featurelabs.com/featuretools-on-spark-2/

Lam, H. T., Minh, T. N., Sinn, M., Buesser, B., & Wistuba, M. (2018). Neural feature learning from relational database. *arXiv preprint arXiv:1801.05372*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436.

L'heureux, A., Grolinger, K., Elyamany, H. F., & Capretz, M. A. (2017). Machine learning with big data: Challenges and approaches. *IEEE Access*, *5*, 7776–7797.

Li, J., & Liu, H. (2017). Challenges of feature selection for big data analytics. *IEEE Intelligent Systems*, *32*(2), 9–15.

Li, J., Zhao, B., Zhang, H., & Jiao, J. (2009). Face recognition system using SVM classifier and feature extraction by PCA and LDA combination. In *2009 international conference on computational intelligence and software engineering* (pp. 1–4).

Lopez, E., & Sartipi, K. (2018). Feature engineering in big data for detection of information systems misuse. In *Proceedings of the 28th annual international conference on computer science and software engineering* (pp. 145–156).

Mocnik, F., Zipf, A., & Fan, H. (2017). Data quality and fitness for purpose. In *Agile conference on geographic information science 2017* (pp. 1–2).

Nanni, L., Ghidoni, S., & Brahnam, S. (2017). Handcrafted vs. non-handcrafted features for computer vision classification. *Pattern Recognition*, *71*, 158–172.

Ng, A. (2013). *Machine learning and AI via brain simulations.* Retrieved 05-05-2019, from http://ai.stanford.edu/~ang/slides/

Pan, M. (2018). *Featuretools4s 0.1.5.* Retrieved 07-02-2020, from https://pypi.org/project/featuretools4s/

Parker, C. (2012). Unexpected challenges in large scale machine learning. In

Proceedings of the 1st international workshop on big data, streams and heterogeneous source mining: Algorithms, systems, programming models and applications (pp. 1–6).

Press, G. (2013). A very short history of Big Data. *Forbes Tech Magazine, May*, *9*.

ProjectPro. (2016). *How data partitioning in Spark helps achieve more parallelism?* Retrieved 07-02-2020, from `\url{https://www.dezyre.com/article/how-data-partitioning-in-spark-helps-achieve-more-parallelism/297}`

Raj, A., & D'Souza, R. (2019). A review on Hadoop eco system for big data.

SAP. (2020). *SAP homepage.* Retrieved 07-02-2020, from `https://www.sap.com/index.html`

Spark, A. (2018). Apache Spark. *Retrieved January*, *17*, 2018.

Van Der Maaten, L., Postma, E., & Van den Herik, J. (2009). Dimensionality reduction: a comparative. *J Mach Learn Res*, *10*(66-71), 13.

Wang, X., & Yu, H. (2005). How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques* (pp. 19–35).

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

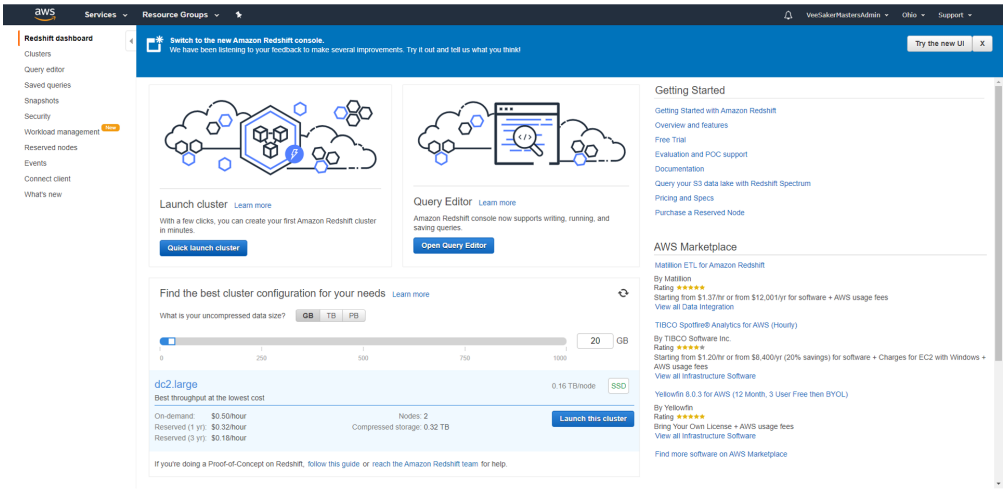
xpanse.ai. (2020). *Xpanse AI homepage.* Retrieved 07-02-2020, from `https://xpanse.ai/`

Appendices

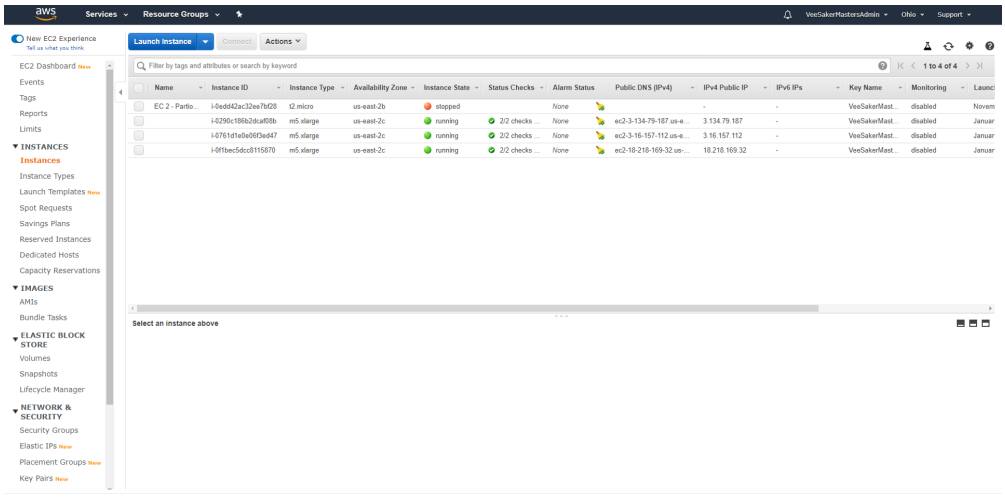
A Appendix: SetUp of Infrastructure

This appendix contains screen shots of the set up of the EC2 and EMR instances.

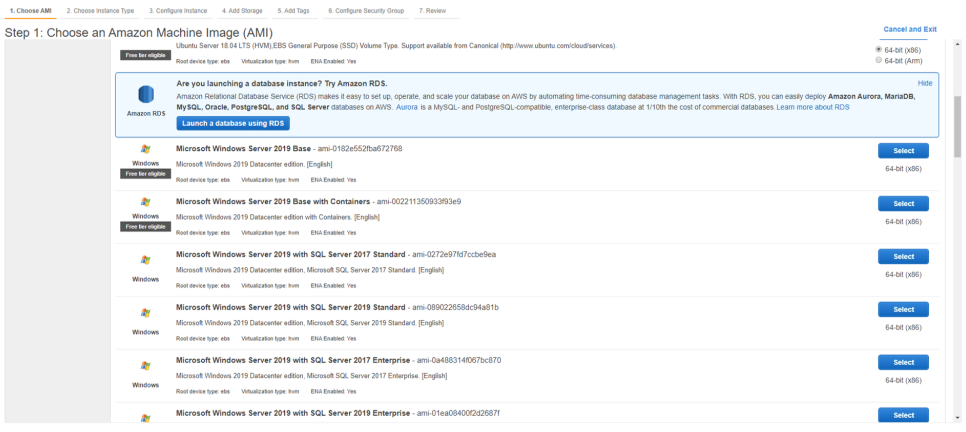
A.1 Setup an EC2 Instance



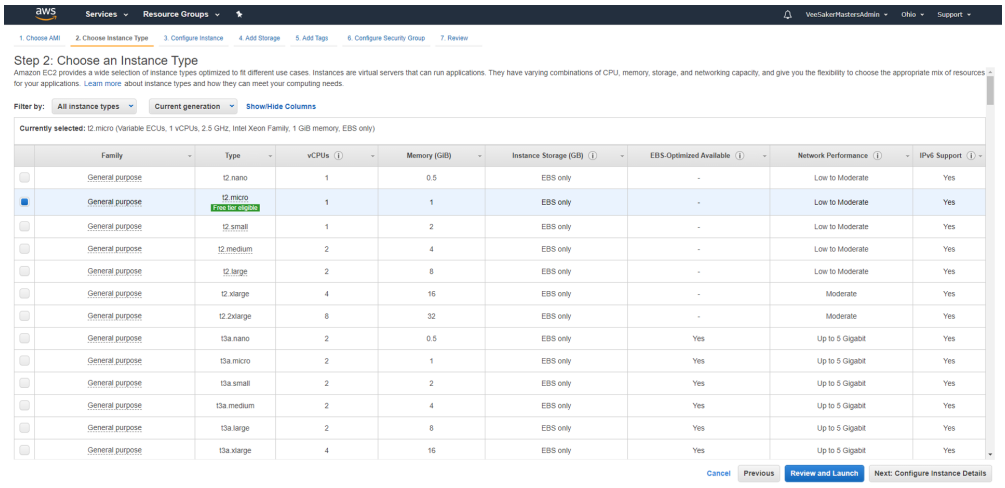
Step One: Welcome Screen for EC Dashboard.



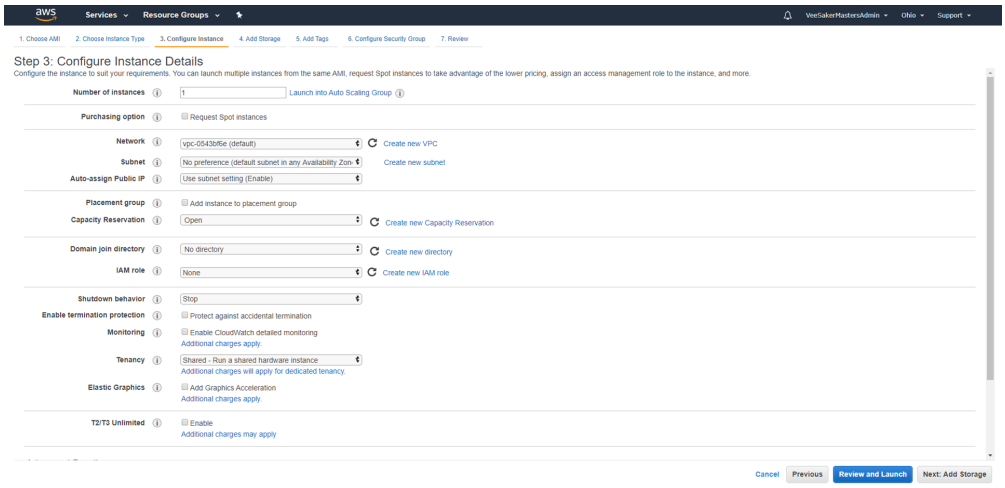
Step Two: Click on EC2 Dashboard to see list of EC2 instances and their status. Make sure you have generated a keypair file for your user by using the sidebar menu under the 'Network & Security' tab - you will need it later so save the file somewhere



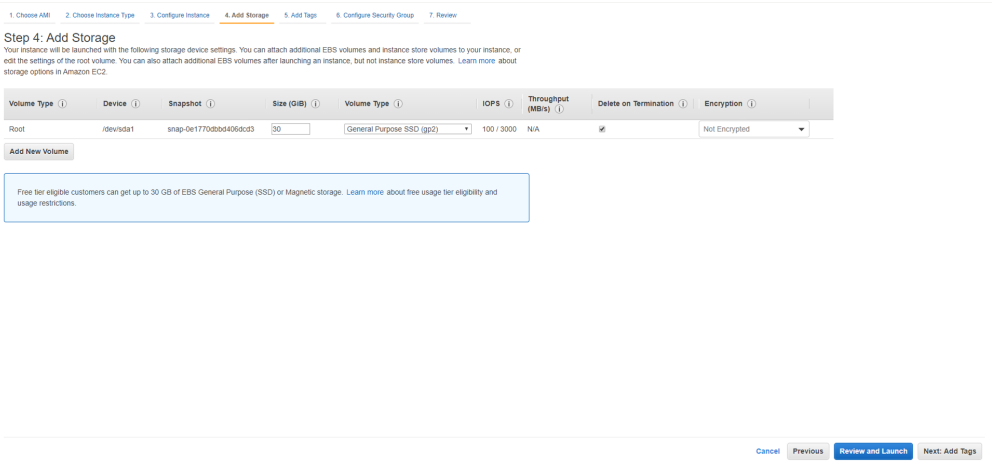
Step Three: Pick an operating system and pre-installed software



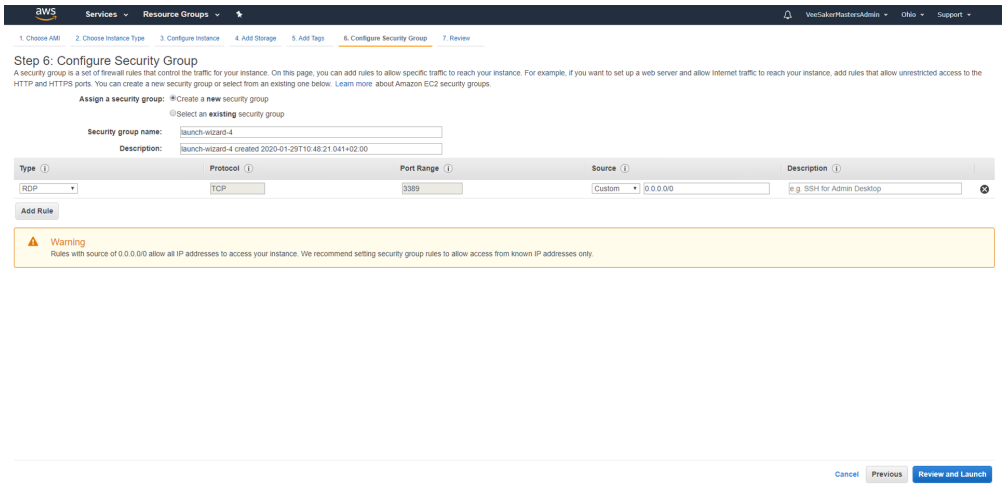
Step Four: Pick an instance type to suit your needs



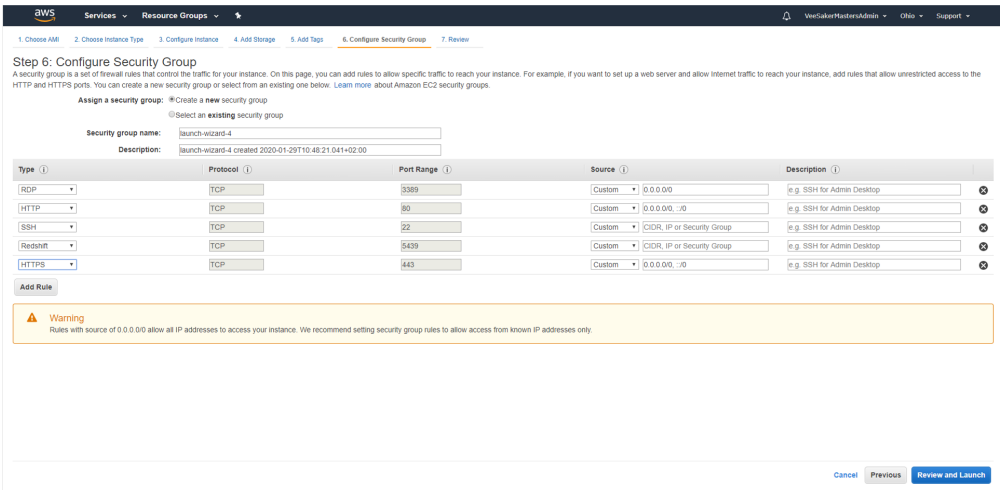
Step Five: Configure the details of your instance including IAM roles etc



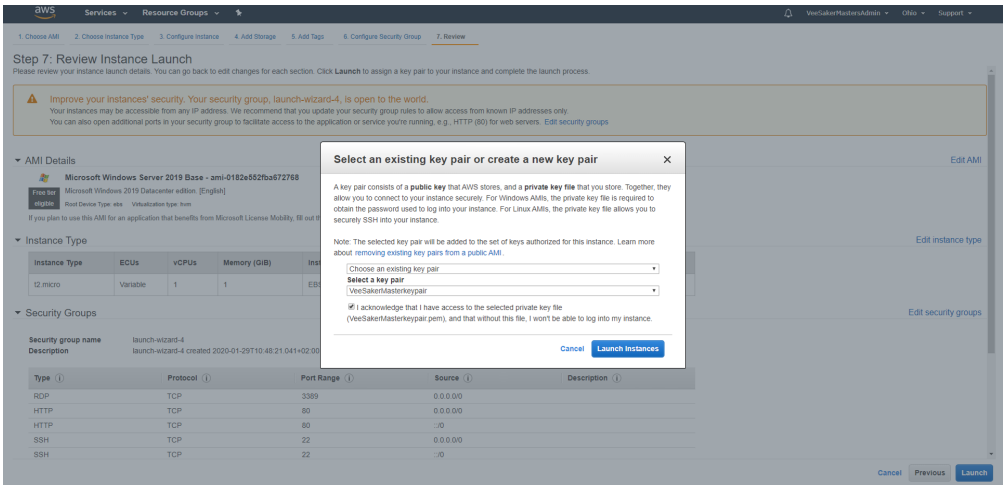
Step Six: Add your storage needs



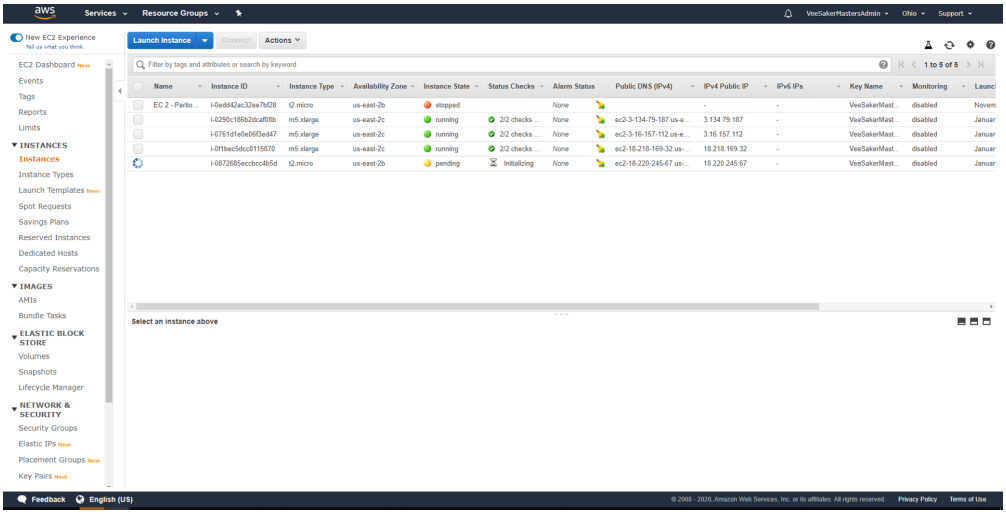
Step Seven: Configure the security group details of your instance



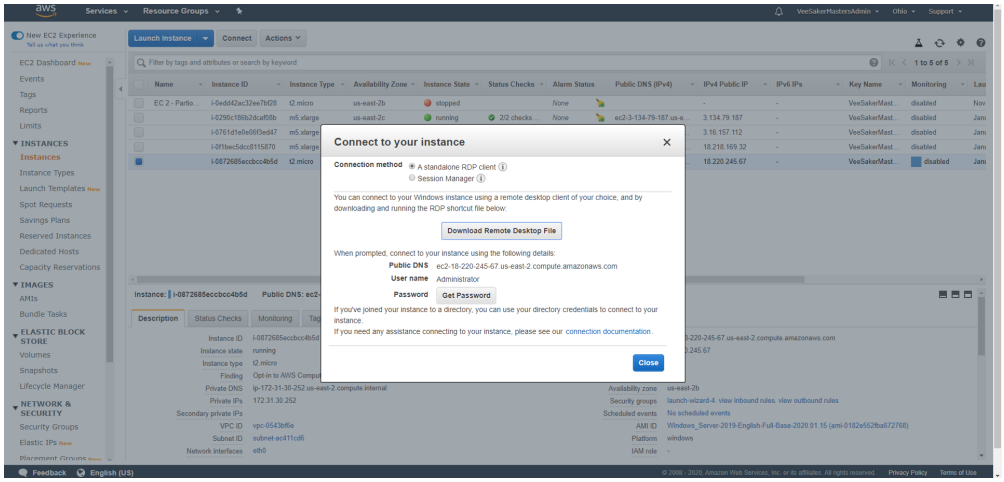
Step Eight: Add ports for SSH, Redshift, HTTP, HTTPS etc.



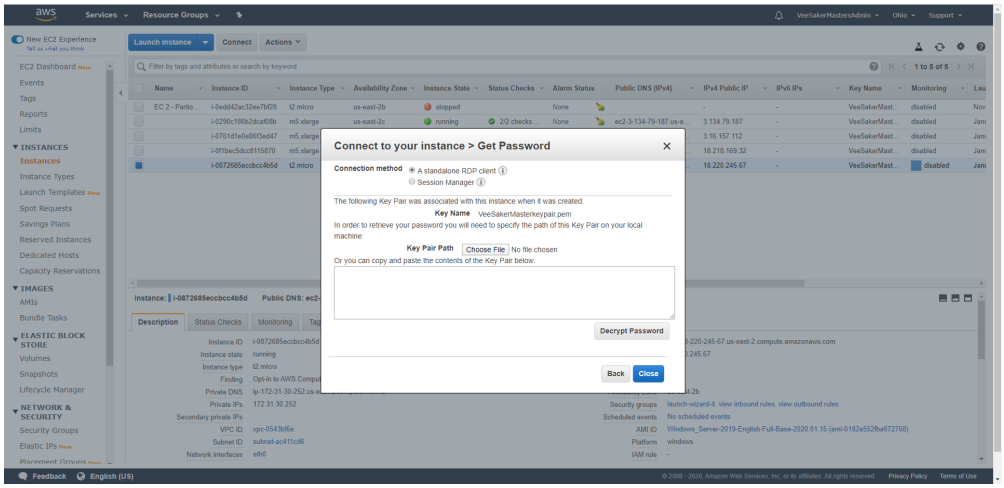
Step Nine: Review your choices and click "Launch". You will be asked to specify an keypair to restrict access to the EC2. Browse to the location that you have save the keypair file and add it. If you do not have one, click cancel and generate one in the side menu bar on the EC2 instance dashboard.



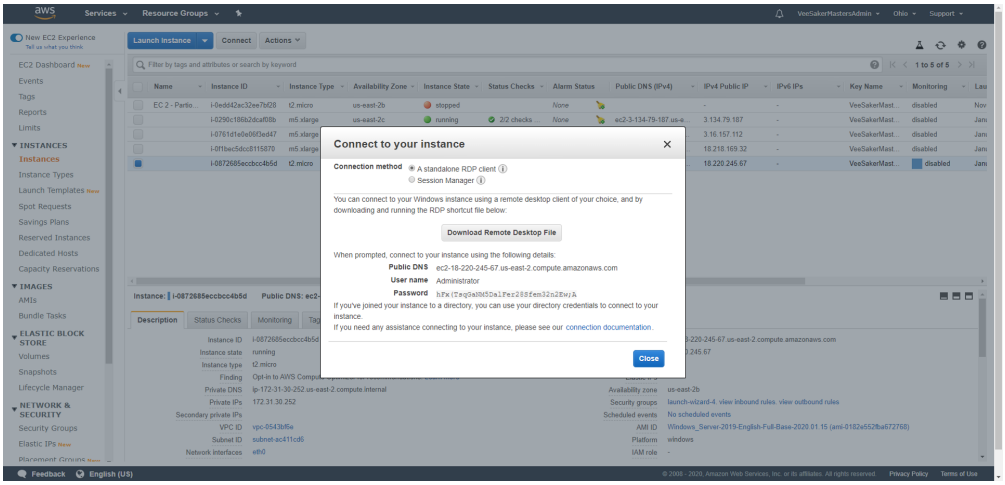
Step Ten: The Dashboard should show a pending status for your new instance



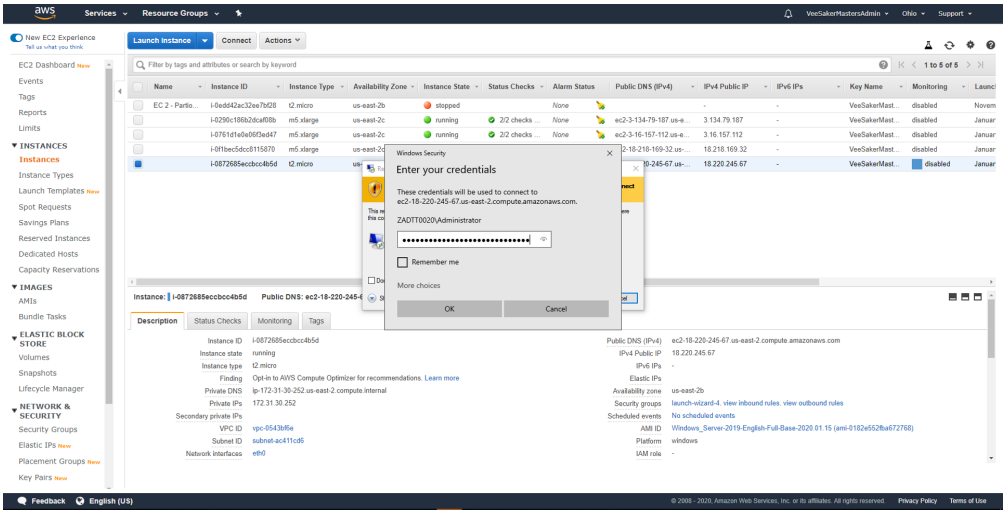
Step Eleven: Once your instance status is ready, click the connect button



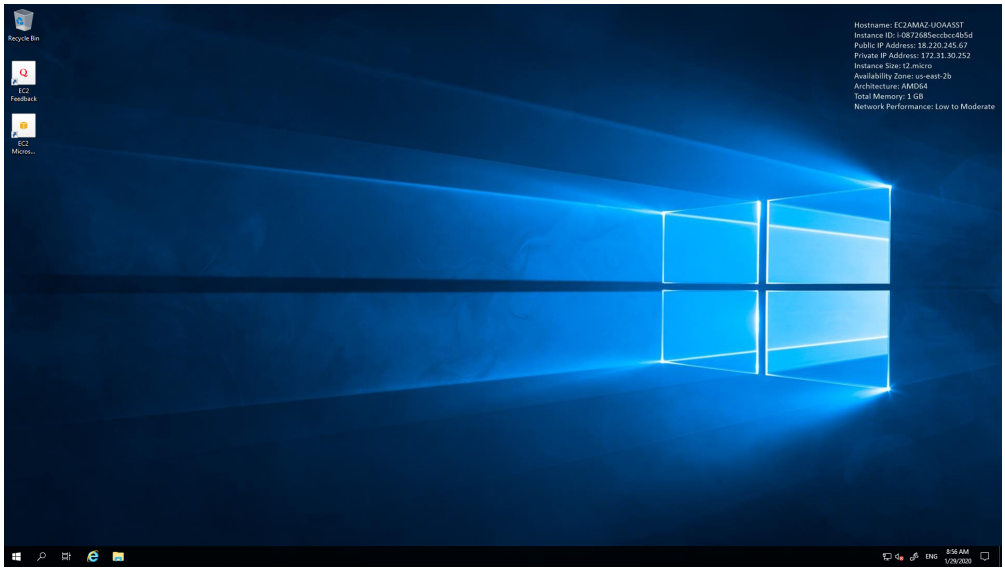
Step Twelve: Click the Get Password button and choose the keypair file



Step Thirteen: Decrypt Password

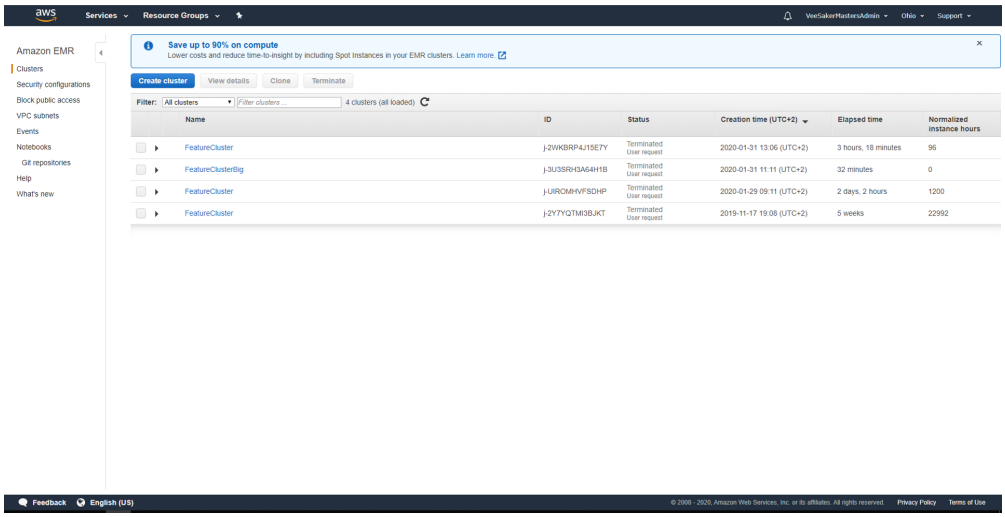


Step Fourteen: Copy the password and use the RDP download to connect to your instance

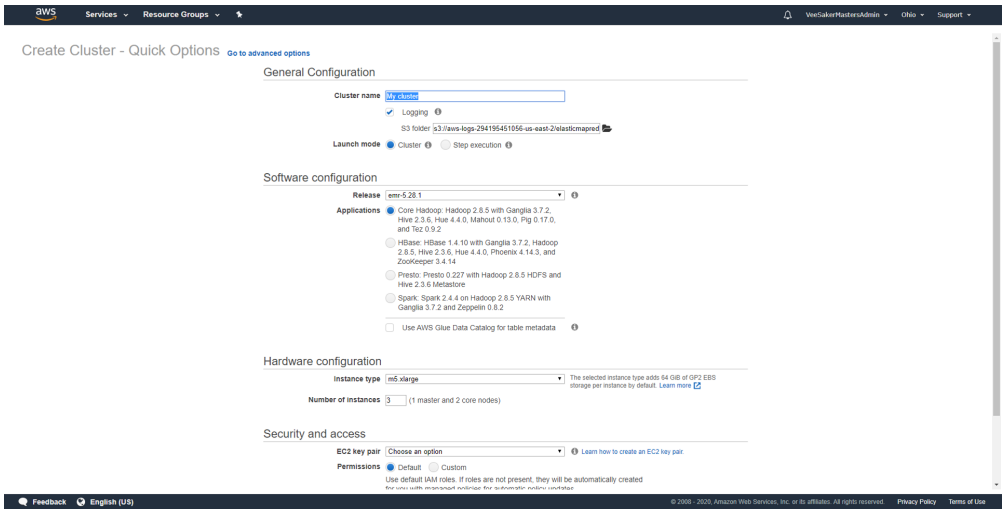


Step Fifteen: The EC2 instance is ready for use

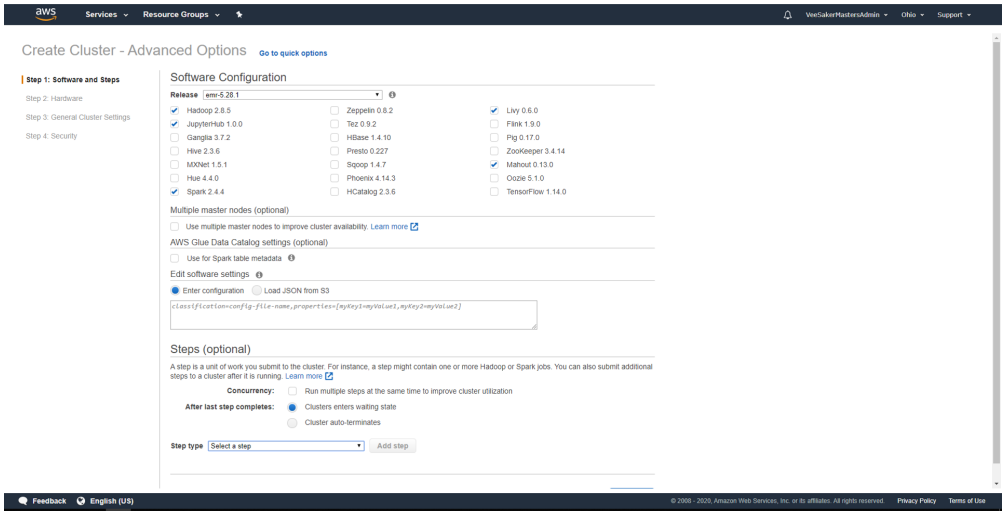
A.2 Setup an AWS EMR Cluster



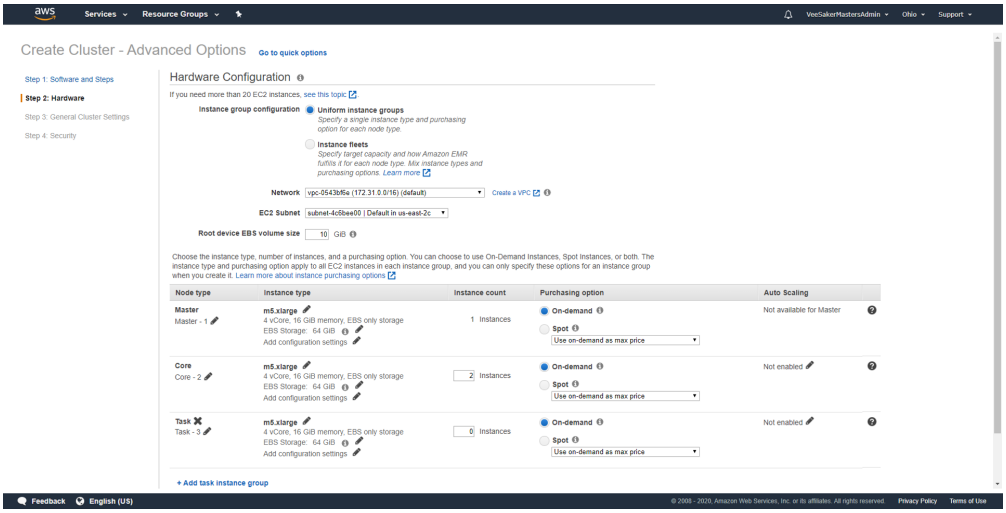
Step One: Navigate to the EMR dashboard and click Create Cluster



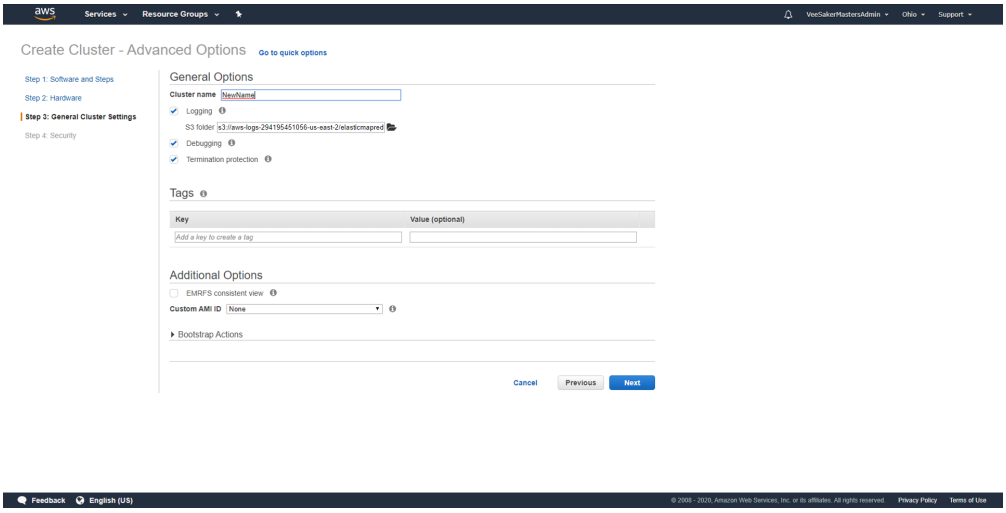
Step Two: Click on the advanced options



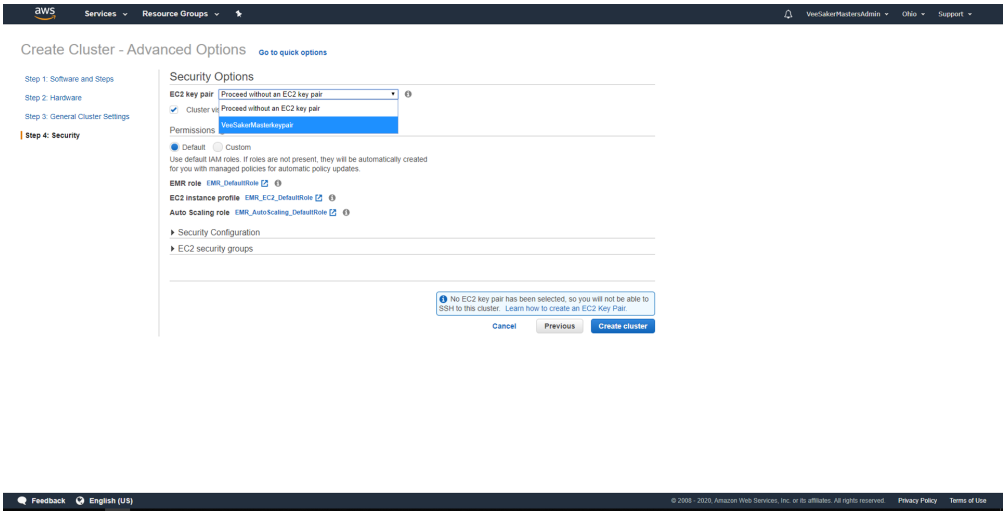
Step Three: Choose software configurations, configuration code etc. if applicable. This is the software configuration for the cluster used in experiment. If you want to be able to use the Notebook environment, make sure you tick on Jupyter Hub option



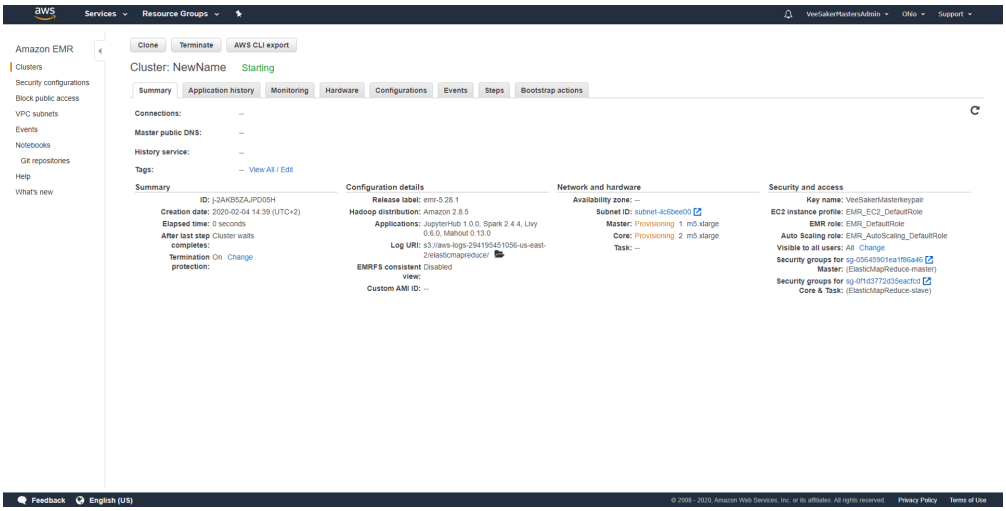
Step Four: On the next page chose the size of your Master and Core nodes. You can add core nodes if you need to here



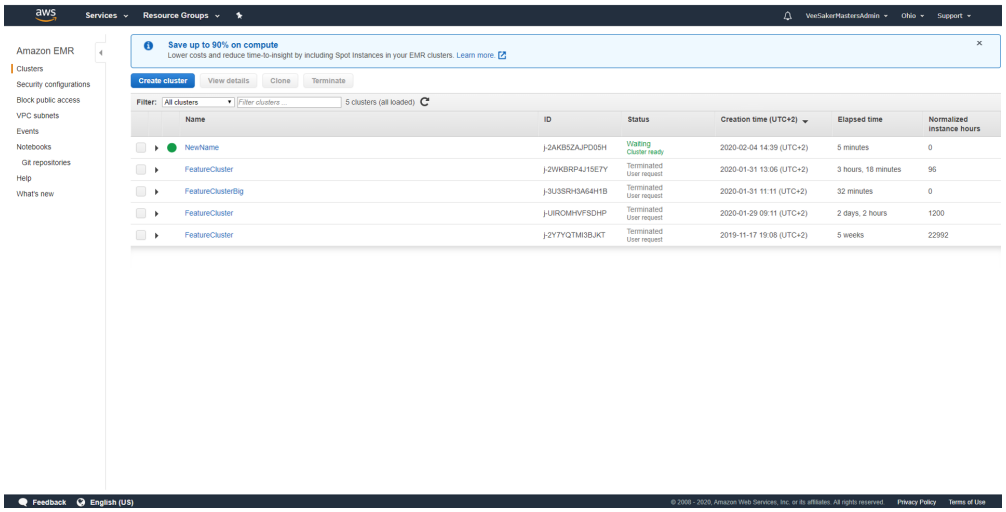
Step Five: Name your node and specify general option i.e. what S3 bucket for logs and scripts and any bootstrap scripts.



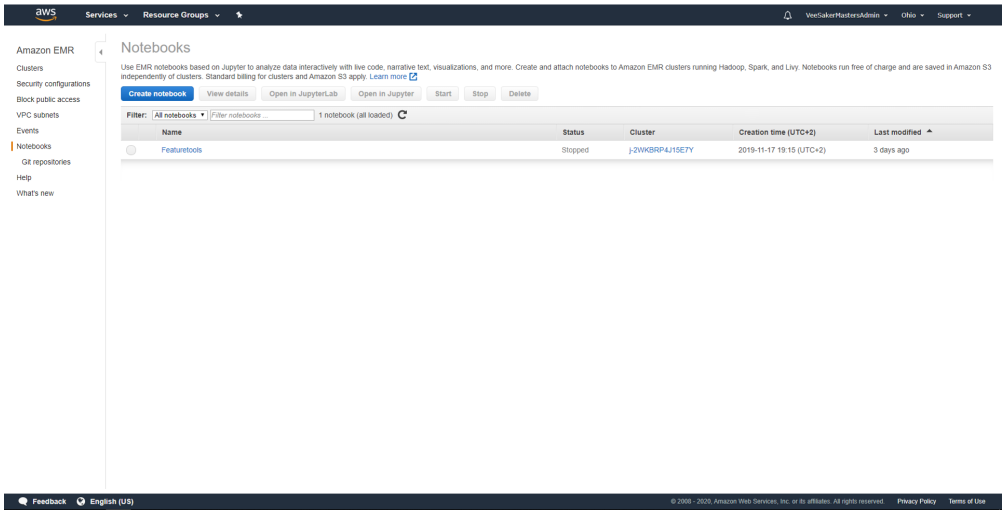
Step Six: Set your security settings i.e. keypair, EC2 security groups etc. If you use a non default security ports, double check your port settings!



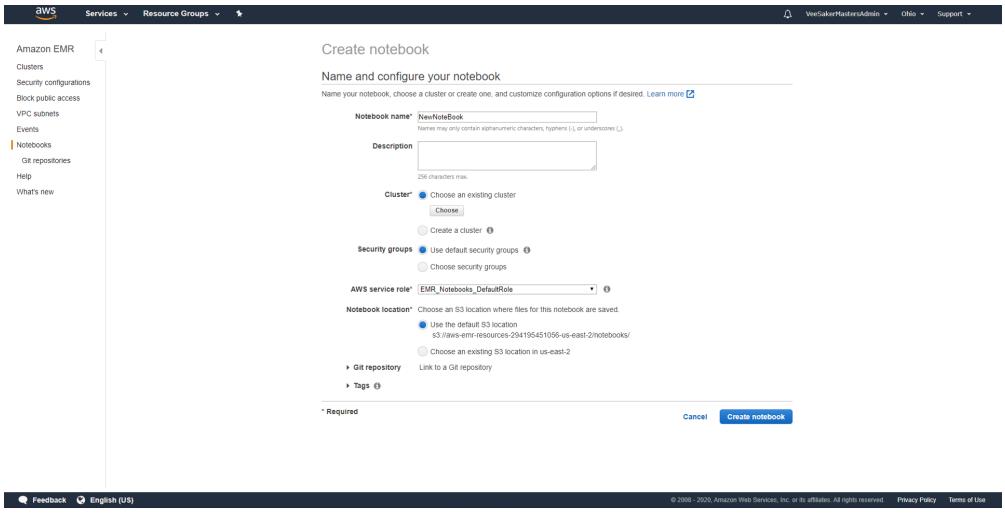
Step Seven: Click Create and the cluster will start setting up



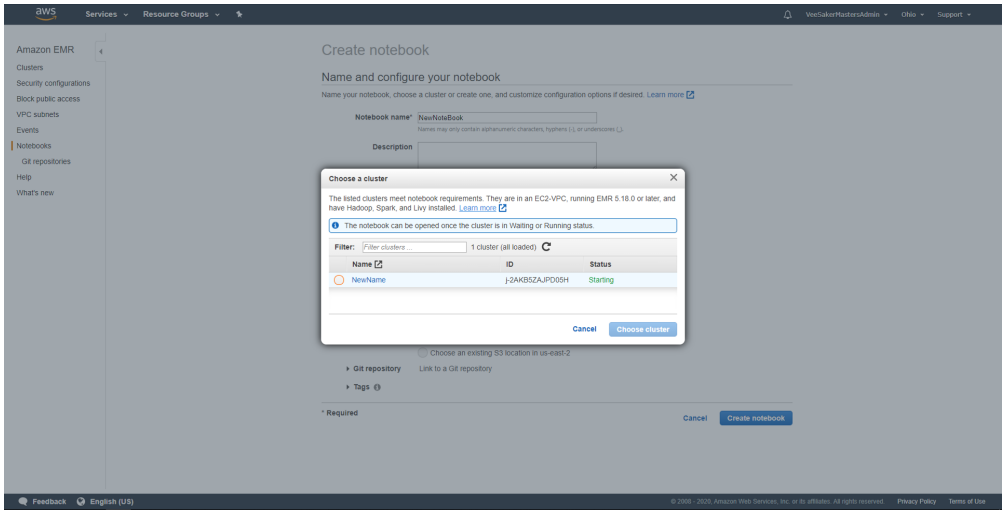
Step Eight: The dashboard will show a 'Starting; status. You can begin to use your cluster when status is 'Waiting'. In the meantime, set up your Notebook environment



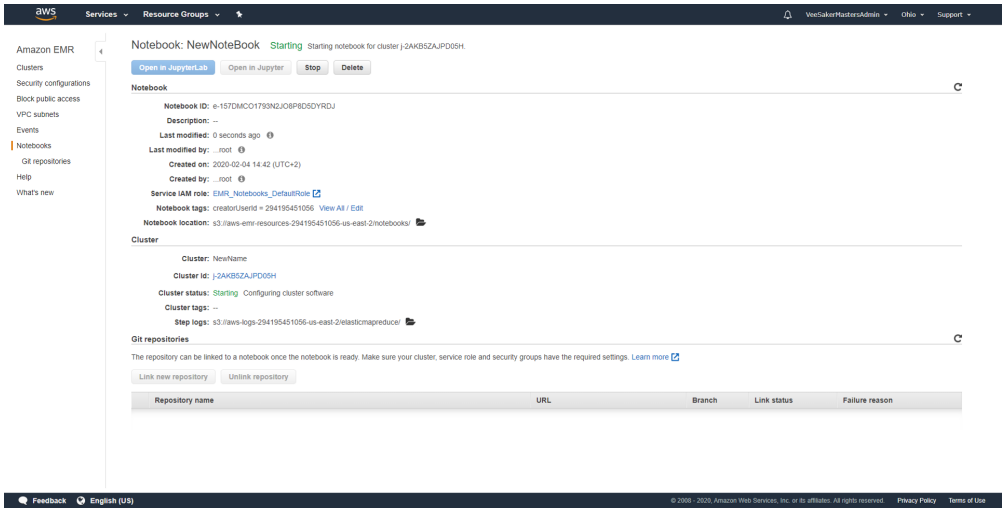
Step Nine: On the Notebook tab, click 'Create Notebook'



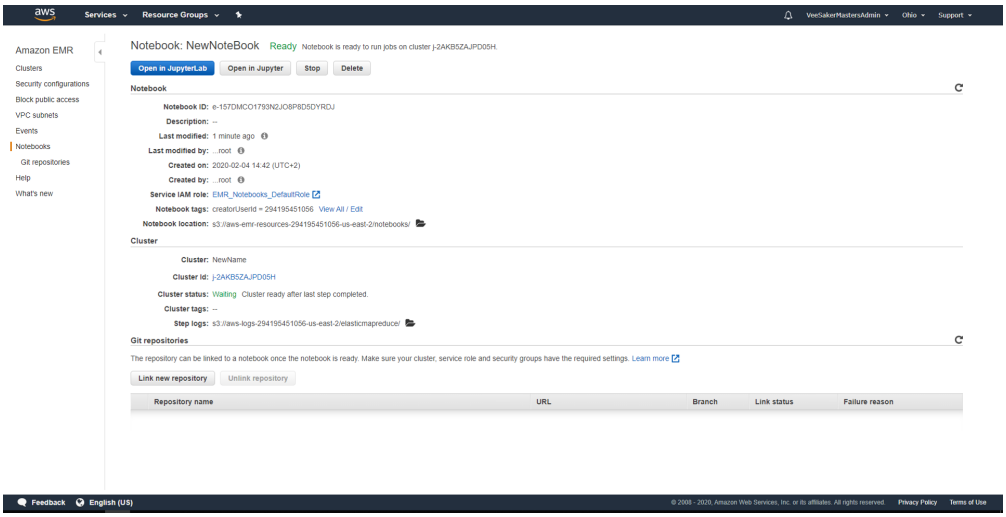
Step Ten: Configure your notebook



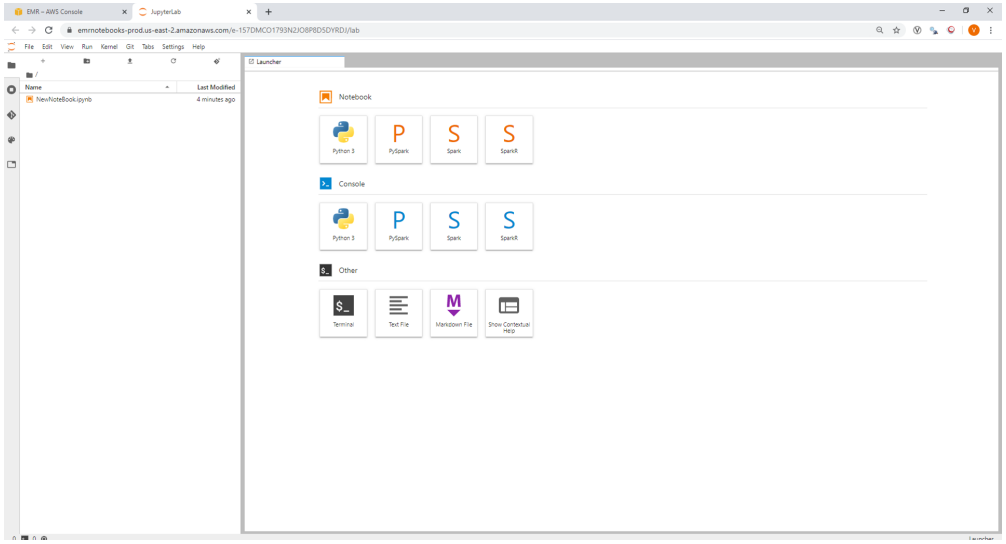
Step Eleven: Choose the cluster you want to which you want to attach your notebook. Only clusters in a "Waiting" or "Starting" are shown. In addition, the cluster must be set up for Jupyter Hub to show or have the correct ports open on a non-default security group



Step Twelve: Your notebook will show a 'Waiting' status until your cluster is ready

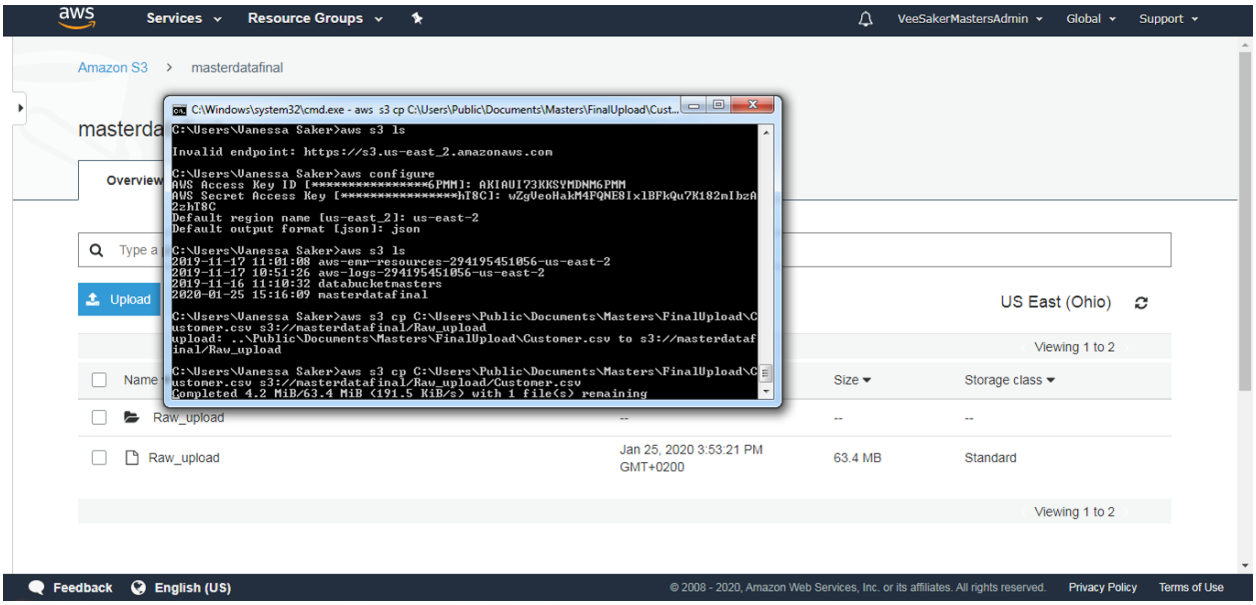


Step Thirteen: Once your cluster is ready, your Notebook will show a 'Ready' Status. Click on the Jupyter Lab button

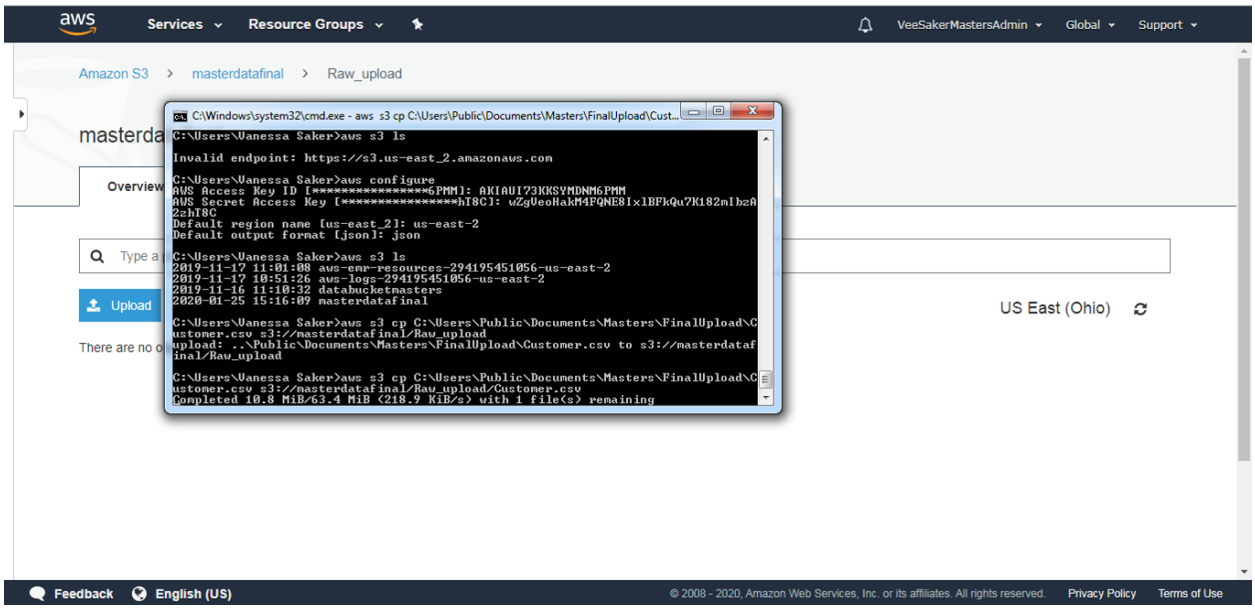


Step Fourteen: The Jupyter Lab environment is available and ready for use

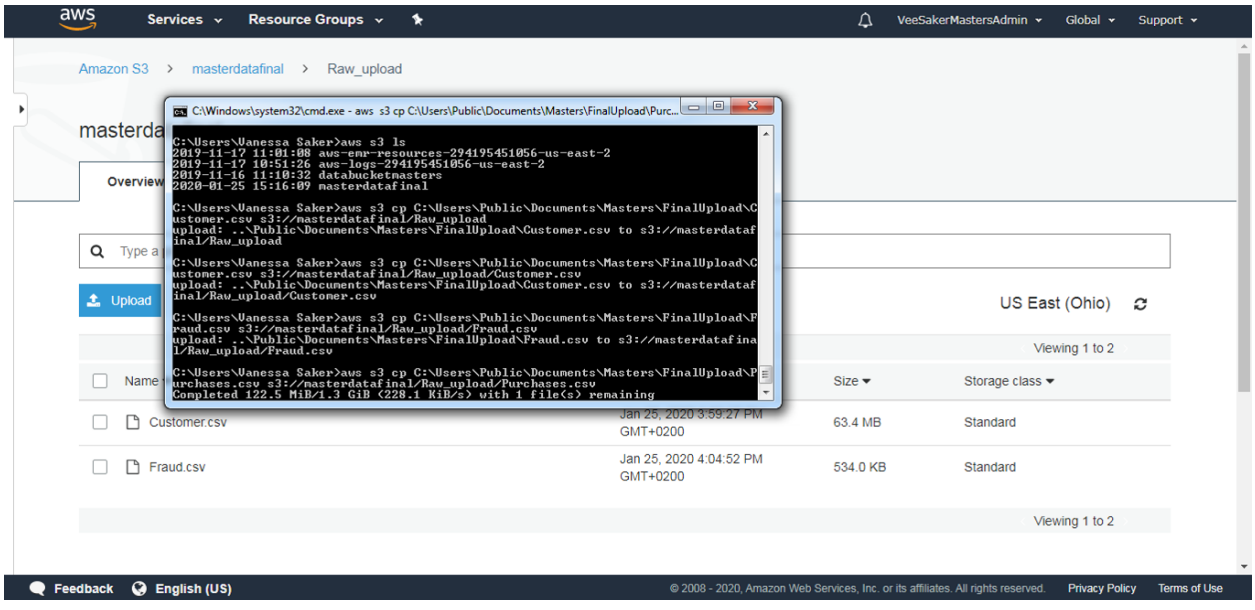
B Appendix: Upload of Data to S3 via AWS CLI



Upload of Customer File



Upload of Fraud File



Upload of Purchase File

C Appendix: Data Load to Redshift Script

```
/****** Script for Loading Data into Redshift ******/
/** Credientials Removed*/


/**Create schema**/
create schema card_data;

/*****Create Tables****/
/**Create Purchase Table**/

--drop table card_data.card_purchases;

CREATE TABLE card_data.card_purchases
(
  TM_PRD_ID varchar(50) NULL,
  CC_AR_ID varchar(50) NULL,
  CST_ID varchar(50) NULL,
  CRD_ID varchar(50) NULL,
  TXN_NO varchar(50) NULL,
  TXN_DATE_CON datetime NULL,
  DATE_RCVD datetime NULL,
  TXN_AMT_ORIG_CCY float NULL,
  TXN_AMT float NULL,
  MRCHT_NO varchar(50) NULL,
  MRCHT_NM varchar(50) NULL,
  MRCHT_CTY_CODE varchar(50) NULL,
  CAMS_MRCHT_CGY_TP_CODE varchar(50) NULL,
  CAMS_MRCHT_CGY_TP_DESC varchar(50) NULL,
  MRCHT_STR_ADR varchar(60) NULL,
  MRCHT_CITY_NM varchar(50) NULL

 );


/**Create Fraud Table**/

CREATE TABLE card_data.card_fraud
(
  CST_ID nvarchar(max) NULL,
  CC_FRD_ID nvarchar(max) NULL,
  CC_FRD_ID_2 nvarchar(max) NULL,
  INVSTGR_NO nvarchar(max) NULL,
  SRC_CRT_TMS datetime NULL,
  LS_INCDT_TMS datetime NULL,
  FRD_EFF_DT datetime NULL,
  FRD_ST_DT datetime NULL,
  CAMS_FRD_ST_TP_CODE nvarchar(255) NULL,
  CAMS_FRD_ST_TP_DESC nvarchar(255) NULL
 );
```

```
/**Create CustomerTable**/

CREATE TABLE card_data.card_customer
(
  CST_ID varchar(50) NULL,
  AGE varchar(50) NULL,
  ANUL_GRS_INCM varchar(50) NULL,
  HOGAN_CST_TP_DESC varchar(50) NULL
 );


/*****Create Tables****/


/**Upload data from S3 file to card_purchases table**/
/**Make empty and blank records null**/
/**Ignore blank lines and fill the record line**/
/**Use a common as a delimiter**/
/**Igonore the first line**/
/**Use the standard time date options**/
/**Remove quotes from text fields**/
/**Escape any problen characters**/
COPY card_data.card_purchases (
TM_PRD_ID
,CC_AR_ID
,CST_ID
,CRD_ID
,TXN_NO
,TXN_DATE_CON
,DATE_RCVD
,TXN_AMT_ORIG_CCY
,TXN_AMT
,MRCHT_NO
,MRCHT_NM
,MRCHT_CTY_CODE
,CAMS_MRCHT_CGY_TP_CODE
,CAMS_MRCHT_CGY_TP_DESC
,MRCHT_STR_ADR
,MRCHT_CITY_NM
 )
from 's3://masterdatafinal/Raw_upload/Purchases.csv'
credentials
    'aws_access_key_id=**;aws_secret_access_key=**zhT8C'
emptyasnull blanksasnull IGNOREBLANKLINES FILLRECORD
delimiter ','
ignoreheader as 1
TIMEFORMAT 'auto'
REMOVEQUOTES
ESCAPE
;

/**Upload data from S3 file to card_fraud table**/
/**Import from a standard CSV file format**/
/**Use a common as a delimiter**/
```

```
/**Make empty and blank records null**/
/**Ignore blank lines and fill the record line**/

COPY card_data.card_fraud (
CST_ID
,CC_FRD_ID
,CC_FRD_ID_2
,INVSTGR_NO
,SRC_CRT_TMS
,LS_INCDT_TMS
,FRD_EFF_DT
,FRD_ST_DT
,CAMS_FRD_ST_TP_CODE
,CAMS_FRD_ST_TP_DESC)
from 's3://masterdatafinal/Raw_upload/Fraud.csv'
credentials
    'aws_access_key_id=**;aws_secret_access_key=**zhT8C'
delimiter ','
csv
ignoreheader as 1
emptyasnull blanksasnull IGNOREBLANKLINES FILLRECORD
;


/**Upload data from S3 file to card_customertable**/
/**Import from a standard CSV file format**/
/**Use a common as a delimiter**/
/**Make empty and blank records null**/
/**Ignore blank lines and fill the record line**/


COPY card_data.card_customer (
  CST_ID,
  AGE,
  ANUL_GRS_INCM,
  HOGAN_CST_TP_DESC
 )
from 's3://masterdatafinal/Raw_upload/Customer.csv'
credentials
    'aws_access_key_id=**;aws_secret_access_key=**zhT8C'
delimiter ','
csv
ignoreheader as 1
emptyasnull blanksasnull IGNOREBLANKLINES FILLRECORD

;
```

D Appendix: Initial SQL Cleaning Script

```
/****** Script for Inital Data Cleaning ******/

/****** 1. Card Purchases ******/
/** Generates a unique key for each transaction in Card
    Purchases table*/

/**Drops the table /**
drop table card_data.card_purchases_unique;


/**Create a new table called card_purchases_unique**/
 CREATE TABLE card_data.card_purchases_unique
( TXN_ID INT IDENTITY(1,1),
  TM_PRD_ID varchar(50) NULL,
  CC_AR_ID varchar(50) NULL,
  CST_ID varchar(50) NULL,
  CRD_ID varchar(50) NULL,
  TXN_NO varchar(50) NULL,
  TXN_DATE_CON datetime NULL,
  DATE_RCVD datetime NULL,
  TXN_AMT_ORIG_CCY varchar(50) NULL,
  TXN_AMT varchar(50) NULL,
  MRCHT_NO varchar(50) NULL,
  MRCHT_NM varchar(50) NULL,
  MRCHT_CTY_CODE varchar(50) NULL,
  CAMS_MRCHT_CGY_TP_CODE varchar(50) NULL,
  CAMS_MRCHT_CGY_TP_DESC varchar(50) NULL,
  MRCHT_STR_ADR varchar(60) NULL,
  MRCHT_CITY_NM varchar(50) NULL
 );

/**Inserts all data from the orginal purchase table with**/
/**Dates between 1 Dec 2015 and 29 Feb 2016**/
/** and Non null customer number**/
 Insert into card_data.card_purchases_unique (
     TM_PRD_ID ,
  CC_AR_ID ,
  CST_ID ,
  CRD_ID ,
  TXN_NO ,
  TXN_DATE_CON ,
  DATE_RCVD ,
  TXN_AMT_ORIG_CCY ,
  TXN_AMT ,
  MRCHT_NO ,
  MRCHT_NM ,
  MRCHT_CTY_CODE,
  CAMS_MRCHT_CGY_TP_CODE ,
  CAMS_MRCHT_CGY_TP_DESC ,
  MRCHT_STR_ADR,
  MRCHT_CITY_NM)
```

```
 select  TM_PRD_ID ,
  CC_AR_ID ,
  CST_ID ,
  CRD_ID ,
  TXN_NO ,
  TXN_DATE_CON ,
  DATE_RCVD ,
  TXN_AMT_ORIG_CCY ,
  TXN_AMT ,
  MRCHT_NO ,
  MRCHT_NM ,
  MRCHT_CTY_CODE,
  CAMS_MRCHT_CGY_TP_CODE ,
  CAMS_MRCHT_CGY_TP_DESC ,
  MRCHT_STR_ADR,
  MRCHT_CITY_NM from card_data.card_purchases
where CST_ID is not null
and TXN_date_con >= '2015-12-01'
and TXN_date_con <= '2016-02-29';



/**Double checks all keys are unqiue**/
 select count(*), TXN_ID from card_data.card_purchases_unique
 group by TXN_ID
 having count(*) >1 ;



 /****** 2. Customer Table ******/
/** DeDups CST_ID for a unique table of customers**/
/** Looks only for "active" clinets **/
/** i.e. clients who have had at least 1 transaction in 3
    month period**/


/**Drops the table /**
drop table card_data.customer_unique;


/**Create a new table called customer_unique**/
CREATE TABLE card_data.customer_unique (
  CST_ID varchar(50) NULL,
  AGE varchar(50) NULL,
  ANUL_GRS_INCM varchar(50) NULL,
  HOGAN_CST_TP_DESC varchar(50) NULL
) ;


/**Inserts all customer data where there is a transaction**/
/**between dates between 1 Dec 2015 and 29 Feb 2016**/
/** and dedupes all cst_id for a single record**/

INSERT INTO card_data.customer_unique
```

```
SELECT
a.CST_ID
,MAX(AGE) AS AGE
, MAX(ANUL_GRS_INCM) AS ANUL_GRS_INCM
, MAX(HOGAN_CST_TP_DESC) AS HOGAN_CST_TP_DESC
from card_data.card_customer a, card_data.card_purchases b
where a.CST_ID = b.CST_ID
and TXN_date_con >= '2015-12-01'
and TXN_date_con <= '2016-02-29'
GROUP BY a.CST_ID
;



/**Double checks all keys are unqiue**/
 select count(*), CST_ID from card_data.customer_unique
 group by CST_ID
 having count(*) >1 ;




/****** 3. Fraud Table ******/
/** DeDups CST_ID for a unique table of fraud customers**/

/**Drops the table /**
drop table card_data.Fraud_unique;

/**Create a new table called card_data.Fraud_unique*/
CREATE TABLE card_data.Fraud_unique(
  CST_ID nvarchar(max) NULL,
  CC_FRD_ID nvarchar(max) NULL,
  CC_FRD_ID_2 nvarchar(max) NULL,
  INVSTGR_NO nvarchar(max) NULL,
  SRC_CRT_TMS datetime NULL,
  LS_INCDT_TMS datetime NULL,
  FRD_EFF_DT datetime NULL,
  FRD_ST_DT datetime NULL,
  CAMS_FRD_ST_TP_CODE nvarchar(255) NULL,
  CAMS_FRD_ST_TP_DESC nvarchar(255) NULL
);



/**Inserts all fraud data where the incident date**/
/**is between 1 Dec 2015 and 03 March 2016**/
/** and dedupes all cst_id for a single record**/
INSERT INTO card_data.Fraud_unique
SELECT
  CST_ID
  ,max(CC_FRD_ID) as CC_FRD_ID
  ,max(CC_FRD_ID_2) as CC_FRD_ID_2
  ,max(INVSTGR_NO) as INVSTGR_NO
```

```
  ,max(SRC_CRT_TMS) as SRC_CRT_TMS
  ,max(LS_INCDT_TMS) as LS_INCDT_TMS
  ,max(FRD_EFF_DT) as FRD_EFF_DT
  ,max(FRD_ST_DT) as FRD_ST_DT
  ,max(CAMS_FRD_ST_TP_CODE) as CAMS_FRD_ST_TP_CODE
  ,max(CAMS_FRD_ST_TP_DESC) as CAMS_FRD_ST_TP_DESC
from card_data.card_fraud
where LS_INCDT_TMS >= '2015-12-01' and LS_INCDT_TMS <=
    '2016-03-07'
GROUP BY CST_ID
;


/**Double checks all keys are unqiue**/
 select count(*), CST_ID from card_data.Fraud_unique
 group by CST_ID
 having count(*) >1 ;



/****** 4. Check size of tables ******/
select count(*) from card_data.Fraud_unique;

select count(*) from card_data.customer_unique ;

select count(*) from card_data.card_purchases_unique ;
```

E Appendix: Partitioning Script

```
#!/usr/bin/env python
# coding: utf-8

# # Packages

# In[]:

#import the packages
import pandas as pd
import hashlib
import os
import psycopg2
import numpy as np
from datetime import datetime
from timeit import default_timer as timer
from sqlalchemy import create_engine
import boto3
#reminder to configure AWS CLI for s3fs to work
import s3fs


# # Redshift Details

# In[]:

#specify redshift details
#specifcs removed
redshift_endpoint =
    "redshiftmasterdatabase.**.us-east-2.redshift.amazonaws.com"
#user inofm and password removed, renenter own
redshift_user = "**********"
redshift_pass = "**********"
port = 5439
dbname = 'dev'


# In[]:

#Gentlemen, start your Redshift engine
engine_string = "postgresql+psycopg2://%s:%s@%s:%d/%s" %
   (redshift_user, redshift_pass, redshift_endpoint, port,
       dbname)
engine = create_engine(engine_string)


# # S3 detials

# In[]:

#aws user infomation. Revoved, user to input own
aws_key = '**************6PMM'
```

```
aws_secret='****************************T8C'
#S3 Bucket information for s3fs package
base_dir = 's3://masterdatafinal/'


# # Number of Paritions

# In[]:

#specify the number of partitions
N_PARTITIONS = 1000


# # Functions
#

# In[]:


def id_to_hash(customer_id):
    """Return a 16-bit integer hash of a customer id string"""
    return
        int(hashlib.md5(customer_id.encode('utf-8')).hexdigest(),
        16)


# In[]:


def write_files_to_partition(df, name, progress = None, affex
    = '.csv'):
    """Partition a dataframe into N_PARTITIONS by hashing the
        id.


    Params
    --------
       df (pandas dataframe):
          dataframe for partition.
          Must have 'CST_ID' column.
       name (str):
        name of dataframe.
        Used for saving the file into paritition.
       progress (int, optional):
        number of rows to be processed before displaying
            information.
                    Defaults to None
       affex = file type to be saved

    Returns:
    --------
       Nothing returned.
   Dataframe is saved as csv files to the parititon in S3 bucket
    """
```

```
    #Starts timer
    start = timer()

    #groups df by the partition number created in hashing
    #iterates thorugh these partitions
    for partition, grouped in df.groupby('part'):
        #specifys the S3 parition and file name
        k = base_dir + f'p{partition}/' + name + affex
        #subsets the df by the parition number
        dfwrite = df[(df['part'] == partition )]
        #write the file the S3 bucket as a csv
        dfwrite = dfwrite.to_csv(None).encode()
        fs = s3fs.S3FileSystem(key=aws_key, secret=aws_secret)
        with fs.open(k, 'wb') as f:
            f.write(dfwrite)



        # Record progress every `progress` steps
        if progress is not None:
            if partition % progress == 0:
                print(f'{100 * round(partition / N_PARTITIONS,
                    2)}% complete. {round(timer() - start)} seconds
                    elapsed.', end = '\r')

    end = timer()
    if progress is not None:
        print(f'\n{df.shape[0]} rows processed in {round(end -
            start)} seconds.')

# In[]:


def make_lables(customer, fraud):
    """Make target labels y or n from reported fraud cases

    Inputs
    --------
     customer: dataset with full set of customer key
     fraud: dataset of reported cases and the date reported

    Returns
    --------
     A list of all customer key with a y or n flag and a date.
     If the flag = Y, the date is date that fraud was reported
     If the flag= N, the date is end of period

     """

     #get all unique customer keys
     unique_customers =
         pd.DataFrame(np.unique(customer["CST_ID"]))
     #give col names to new dataset
     unique_customers.columns = ["CST_ID"]
```

```
    #merge with Fraud table - match on key and keep incident
        date
    label_tabel = pd.merge(unique_customers,
              fraud[['CST_ID', 'LS_INCDT_TMS']],
              on='CST_ID', how = 'left')
    # Input last date of time period
    date =
        pd.to_datetime(np.datetime64(datetime.strptime("2016-02-29",
        "%Y-%m-%d")))
    #create table with flag and dates
    label_tabel["date"] = date
    label_tabel["FRD_FLG"] =
        np.where(label_tabel["LS_INCDT_TMS"].isnull(), 'N', 'Y')
    label_tabel["CUT_OFF_TIME"] =
        np.where(label_tabel["LS_INCDT_TMS"].isnull(),
        label_tabel["date"], label_tabel["LS_INCDT_TMS"])
    label_tabel["CUT_OFF_TIME"] =
        pd.to_datetime(label_tabel["CUT_OFF_TIME"])

    #drop unused variables
    final = label_tabel[["CST_ID","FRD_FLG","CUT_OFF_TIME"]]
    return final



# # Read in Data from Redshift

# In[]:

#read in fraud data
fraud = pd.read_sql_query('SELECT * FROM
    card_data.fraud_unique;', engine)


# In[9]:

#read in customer data
customer = pd.read_sql_query('SELECT * FROM
    card_data.customer_unique;', engine)


# In[10]:

#read in purchases data
purchases = pd.read_sql_query('SELECT * FROM
    card_data.card_purchases_unique;', engine)


# # Make Labels

# In[]:

#make the lables from fraud and customer data
fraud = make_lables(customer, fraud)
```

```
# # Apply the Hash Algorithm

# In[]:


#fraud
#convert cst_id to a string and then hash
fraud["cst_id_str"] = fraud["cst_id"].astype(str)
fraud["part"] = fraud["cst_id_str"].apply(id_to_hash) %
    N_PARTITIONS


# In[]:


#customer
#convert cst_id to a string and then hash
customer["cst_id_str"] = customer["cst_id"].astype(str)
customer["part"] = customer["cst_id_str"].apply(id_to_hash) %
    N_PARTITIONS


# In[ ]:


#purcashes
#convert cst_id to a string and then hash
purchases["cst_id_str"] = purchases["cst_id"].astype(str)
purchases["part"] = purchases["cst_id_str"].apply(id_to_hash)
    % N_PARTITIONS


# # Write to S#

# In[]:

#parition and write fraud file
write_files_to_partition(fraud, "fraud", progress = 10)


# In[]:

#parition and write customer file
write_files_to_partition(customer, "customer", progress = 10)


# In[ ]:

#parition and write purchases file
write_files_to_partition(purchases, "purchases", progress = 10)
```

F Appendix: Feature Engineering and Preprocessing on a Single Partition

```
#!/usr/bin/env python
# coding: utf-8

# Clean Up and Feature Engineering


# ## Install Packages

# In[]:

# install pandas, featuretools, numpy, datetime
import pandas as pd
import featuretools as ft
import numpy as np
from datetime import datetime
import featuretools.variable_types as vtypes



# ## Import Data
#

# In[]:

#Import Fraud data from subfile p0
#parse dates and category types on import
fraud = pd.read_csv('./p0/fraud.csv',
              parse_dates=['CUT_OFF_TIME'],
               infer_datetime_format = True)


# In[]:


#Import Purchases data from subfile p0
#parse dates and category types on import
purchases = pd.read_csv('./p0/purchases.csv',
              parse_dates=['TXN_DATE_CON', 'DATE_RCVD'],
               infer_datetime_format = True,
               dtype = {'CC_AR_ID':'category',
                      "CRD_ID": 'category','MRCHT_NO':
                          'category',
                      'MRCHT_NM':'category',
                      'MRCHT_CTY_CODE':'category',
                      'CAMS_MRCHT_CGY_TP_CODE':'category',
                      'CAMS_MRCHT_CGY_TP_DESC':'category',
                      'MRCHT_STR_ADR':'category',
                      'MRCHT_CITY_NM':'category',
                     })
```

```
# In[]:

#Import customer data from subfile p0
#parse dates and category types on import
customer = pd.read_csv('./p0/customer.csv',
          infer_datetime_format = True,
                dtype = {'AGE':
                    'category','HOGAN_CST_TP_DESC':'category'})


# ## Clean up and preprocess



# In[]:


def customer_preprocess(customer):
    """Impute Missing Data from cutomer tables and drop unused
        variables

    Inputs
    ---------
    customer = raw cutomer table

    Returns
    ---------
    Imputed and cleaned customer table

    """
    #impute income with mean
    customer['ANUL_GRS_INCM'] =
        customer['ANUL_GRS_INCM'].fillna((customer['ANUL_GRS_INCM'].mean()))
    # impute age with mode
    customer['AGE'] =
        customer['AGE'].fillna((customer['AGE'].mode()))
    # impute hogan type with mode
    customer['HOGAN_CST_TP_DESC'] =
        customer['HOGAN_CST_TP_DESC'].fillna((customer['HOGAN_CST_TP_DESC'].mode()))


    #keeps clean variables
    customer = customer[["CST_ID","AGE","ANUL_GRS_INCM",
        "HOGAN_CST_TP_DESC"]]

    return customer


# In[]:
```

```
def purchases_clean(df):
    """Impute Missing Data from purchase table and drop unused
        variables

    Inputs
    ---------
    df = raw purchases table

    Returns
    ---------
    Imputed and cleaned purchase table

    """
    # if mrcht_cty_code does not have a category unkown, make
        one and the use to impute null values
    # if it does have one, use it to imute null values
    if 'UKN' in purchases['MRCHT_CTY_CODE'].cat.categories:
        df['MRCHT_CTY_CODE'] = df['MRCHT_CTY_CODE'].fillna("UKN")
    else:
        df['MRCHT_CTY_CODE'] =
            df['MRCHT_CTY_CODE'].cat.add_categories('UKN')
        df['MRCHT_CTY_CODE'] = df['MRCHT_CTY_CODE'].fillna("UKN")

    #impute missing category codes to the mode
    df['CAMS_MRCHT_CGY_TP_CODE'] =
        df['CAMS_MRCHT_CGY_TP_CODE'].fillna((df['CAMS_MRCHT_CGY_TP_CODE'].mode()))

    # if mrcht_cite name does not have a category unkown, make
        one and the use to impute null values
    # if it does have one, use it to imute null values
    if 'UKN' in purchases['MRCHT_CTY_CODE'].cat.categories:
        df['MRCHT_CITY_NM'] = df['MRCHT_CITY_NM'].fillna("UKN")
    else:
        df['MRCHT_CITY_NM'] =
            df['MRCHT_CITY_NM'].cat.add_categories('UKN')
        df['MRCHT_CITY_NM'] = df['MRCHT_CITY_NM'].fillna("UKN")

    #keep only certain variables

    df = df[["TXN_ID",
             "CC_AR_ID",
             "CST_ID",
             "CRD_ID",
             "TXN_DATE_CON",
             "TXN_AMT",
             "MRCHT_NM",
             "MRCHT_CTY_CODE",
             "CAMS_MRCHT_CGY_TP_CODE",
             "MRCHT_CITY_NM"]]
    return df
```

```
# ## Feature Engneering

# In[]:


def feature_engineering(customer,fraud,purchases):
    """Creates feature matrix

    Inputs
    ---------
    customer = cleaned and imouted file of all customer -
        target entity
    fraud = label table containing date of reported fraud and
        target variable
    purchases = cleaned and imputed purchase table

    Returns
    ---------
    Feature matrix and list of features produced

    """


    #creates enity set called
    es = ft.EntitySet(id = 'fraud')
    # adds customer dataframe to entity set and specifes index
        cst_id
    # specifies which varibales are categories
    es.entity_from_dataframe(entity_id='customers',
        dataframe=customer,
                    index = 'CST_ID',
                    variable_types = {'AGE': vtypes.Categorical,
                                  'HOGAN_CST_TP_DESC':
                                      vtypes.Categorical})

    # adds purchases dataframe to entity set and specifes index
        txn_id and time index TXN_date_con
    # specifies which varibales are categories
    es.entity_from_dataframe(entity_id='purchases',
        dataframe=purchases,
                    index = 'TXN_ID',
                    time_index = 'TXN_DATE_CON',
                    variable_types = {"CC_AR_ID":
                        vtypes.Categorical,
                                  "CRD_ID": vtypes.Categorical,
                       'MRCHT_NM':vtypes.Categorical,
                       'MRCHT_CTY_CODE':vtypes.Categorical,
                       'CAMS_MRCHT_CGY_TP_CODE':vtypes.Categorical,
                       'MRCHT_CITY_NM':vtypes.Categorical })

    #defines how customer and purcahse tables are related
    r_purchases = ft.Relationship(es['customers']['CST_ID'],
        es['purchases']['CST_ID'])
```

```
    #adds the realtionship to entity set
    es.add_relationships([r_purchases])

    #defines a cut off time from the fraud data
    cutoff_time = fraud[["CST_ID","CUT_OFF_TIME"]]

    #defines transformative primitives
    trans_primitives = ['is_weekend',"weekday", 'cum_sum',
        'day', 'month', 'diff', 'time_since_previous']
    #defines aggreagtion primitives
    agg_primitives = ['sum', 'time_since_last',
        'avg_time_between', 'mode', 'num_unique', 'min',
              'mean', 'max', 'std', 'count']
    #rund the dfs algorithm with specfied cutt-off times,
        primititeve and target entitity
    feature_matrix, feature_defs = ft.dfs(entityset=es,
        target_entity='customers',
                    cutoff_time =cutoff_time,
                    agg_primitives = agg_primitives,
                    trans_primitives = trans_primitives,
                    max_depth=2, features_only=False)
    return (feature_matrix, feature_defs)


# ## Run functions
#



# In[]:

#Clean the customer table
customer = customer_preprocess(customer)
#Clean purchases table
purchases= purchases_clean(purchases)
#create feature matrix
feature_matrix_out, feature_defs_out =
    feature_engineering(customer,fraud_labels,purchases)


# In[]:

#Inspect output table
feature_matrix_out
```

G Appendix: Full List of Primitives available in Featuretools

Num	Name	Type	Description
0	n_most_common	aggregation	Determines the 'n' most common elements.
1	percent_true	aggregation	Determines the percent of 'True' values.
2	num_true	aggregation	Counts the number of 'True' values.
3	mean	aggregation	Computes the average for a list of values.
4	min	aggregation	Calculates the smallest value, ignoring 'NaN' values.
5	all	aggregation	Calculates if all values are 'True' in a list.
6	avg_time_between	aggregation	Computes the average number of seconds between consecutive events.
7	time_since_first	aggregation	Calculates the time elapsed since the first datetime (in seconds).
8	first	aggregation	Determines the first value in a list.
9	skew	aggregation	Computes the extent to which a distribution differs from a normal distribution.
10	mode	aggregation	Determines the most commonly repeated value.
11	sum	aggregation	Calculates the total addition, ignoring 'NaN'.
12	count	aggregation	Determines the total number of values, excluding 'NaN'.
13	max	aggregation	Calculates the highest value, ignoring 'NaN' values.
14	any	aggregation	Determines if any value is 'True' in a list.
15	num_unique	aggregation	Determines the number of distinct values, ignoring 'NaN' values.
16	median	aggregation	Determines the middlemost number in a list of values.
17	std	aggregation	Computes the dispersion relative to the mean value, ignoring 'NaN'.
18	last	aggregation	Determines the last value in a list.
19	trend	aggregation	Calculates the trend of a variable over time.
20	entropy	aggregation	Calculates the entropy for a categorical variable
21	time_since_last	aggregation	Calculates the time elapsed since the last datetime (default in seconds).
22	num_words	transform	Determines the number of words in a string by counting the spaces.
23	second	transform	Determines the seconds value of a datetime.
24	day	transform	Determines the day of the month from a datetime.
25	add_numeric	transform	Element-wise addition of two lists.
26	modulo_numeric	transform	Element-wise modulo of two lists.
27	cum_mean	transform	Calculates the cumulative mean.
28	greater_than_scalar	transform	Determines if values are greater than a given scalar.
29	greater_than	transform	Determines if values in one list are greater than another list.
30	equal_scalar	transform	Determines if values in a list are equal to a given scalar.
31	weekday	transform	Determines the day of the week from a datetime.
32	latitude	transform	Returns the first tuple value in a list of LatLong tuples.
33	absolute	transform	Computes the absolute value of a number.
34	or	transform	Element-wise logical OR of two lists.
35	less_than_scalar	transform	Determines if values are less than a given scalar.
36	modulo_numeric_scalar	transform	Return the modulo of each element in the list by a scalar.
37	cum_sum	transform	Calculates the cumulative sum.
38	multiply_numeric	transform	Element-wise multiplication of two lists.
39	equal	transform	Determines if values in one list are equal to another list.
40	multiply_boolean	transform	Element-wise multiplication of two lists of boolean values.
41	cum_min	transform	Calculates the cumulative minimum.
42	less_than_equal_to_scalar	transform	Determines if values are less than or equal to a given scalar.
43	num_characters	transform	Calculates the number of characters in a string.
44	negate	transform	Negates a numeric value.
45	month	transform	Determines the month value of a datetime.
46	divide_numeric_scalar	transform	Divide each element in the list by a scalar.
47	less_than	transform	Determines if values in one list are less than another list.
48	percentile	transform	Determines the percentile rank for each value in a list.
49	diff	transform	Compute the difference between the value in a list and the
50	and	transform	Element-wise logical AND of two lists.
51	time_since_previous	transform	Compute the time since the previous entry in a list.
52	multiply_numeric_scalar	transform	Multiply each element in the list by a scalar.
53	time_since	transform	Calculates time from a value to a specified cutoff datetime.
54	is_weekend	transform	Determines if a date falls on a weekend.
55	is_null	transform	Determines if a value is null.
56	greater_than_equal_to	transform	Determines if values in one list are greater than or equal to another list.
57	add_numeric_scalar	transform	Add a scalar to each value in the list.
58	divide_numeric	transform	Element-wise division of two lists.
59	cum_count	transform	Calculates the cumulative count.
60	not_equal_scalar	transform	Determines if values in a list are not equal to a given scalar.
61	isin	transform	Determines whether a value is present in a provided list.
62	haversine	transform	Calculates the approximate haversine distance between two LatLong
63	cum_max	transform	Calculates the cumulative maximum.
64	scalar_subtract_numeric_feature	transform	Subtract each value in the list from a given scalar.
65	not_equal	transform	Determines if values in one list are not equal to another list.
66	divide_by_feature	transform	Divide a scalar by each value in the list.
67	modulo_by_feature	transform	Return the modulo of a scalar by each element in the list.
68	subtract_numeric	transform	Element-wise subtraction of two lists.
69	longitude	transform	Returns the second tuple value in a list of LatLong tuples.
70	year	transform	Determines the year value of a datetime.
71	minute	transform	Determines the minutes value of a datetime.
72	less_than_equal_to	transform	Determines if values in one list are less than or equal to another list.
73	not	transform	Negates a boolean value.
74	week	transform	Determines the week of the year from a datetime.
75	hour	transform	Determines the hour value of a datetime.
76	greater_than_equal_to_scalar	transform	Determines if values are greater than or equal to a given scalar.
77	subtract_numeric_scalar	transform	Subtract a scalar from each element in the list.

H Appendix: PySpark Code in Notebook for Scaling

```
#Start SparkSession
print('Start Spark')

#Install featuretools
sc.install_pypi_package("featuretools")

#Install s3fs
sc.install_pypi_package("s3fs")

#Install datetime package
sc.install_pypi_package("datetime")

#Check packages are correctly installed
sc.list_packages()


import pandas as pd
import numpy as np
import s3fs
import featuretools as ft
from datetime import datetime
import featuretools.variable_types as vtypes


#AWS user info masked. USer to inoutput own
aws_key ='*****************6PMM'
aws_secret = '*********************************hT8C'
#specify S3 bucket file route
fs = s3fs.S3FileSystem(key=aws_key, secret=aws_secret)

def customer_preprocess(customer):
    """Impute Missing Data from cutomer tables and drop unused
        variables

    Inputs
    ---------
    customer = raw cutomer table

    Returns
    ---------
    Imputed and cleaned customer table

    """
    #impute income with mean
    customer['ANUL_GRS_INCM'] =
        customer['ANUL_GRS_INCM'].fillna((customer['ANUL_GRS_INCM'].mean()))
    # impute age with mode
    customer['AGE'] =
        customer['AGE'].fillna((customer['AGE'].mode()))
```

```
    # impute hogan type with mode
    customer['HOGAN_CST_TP_DESC'] =
        customer['HOGAN_CST_TP_DESC'].fillna((customer['HOGAN_CST_TP_DESC'].mode()))


    #keeps clean variables
    customer = customer[["CST_ID","AGE","ANUL_GRS_INCM",
        "HOGAN_CST_TP_DESC"]]

    return customer

def purchases_clean(df):
    """Impute Missing Data from purchase table and drop unused
        variables

    Inputs
    ---------
    df = raw purchases table

    Returns
    ---------
    Imputed and cleaned purchase table

    """
    # if mrcht_cty_code does not have a category unkown, make
        one and the use to impute null values
    # if it does have one, use it to imute null values
    if 'UKN' in purchases['MRCHT_CTY_CODE'].cat.categories:
       df['MRCHT_CTY_CODE'] = df['MRCHT_CTY_CODE'].fillna("UKN")
    else:
       df['MRCHT_CTY_CODE'] =
           df['MRCHT_CTY_CODE'].cat.add_categories('UKN')
       df['MRCHT_CTY_CODE'] = df['MRCHT_CTY_CODE'].fillna("UKN")

    #impute missing category codes to the mode
    df['CAMS_MRCHT_CGY_TP_CODE'] =
        df['CAMS_MRCHT_CGY_TP_CODE'].fillna((df['CAMS_MRCHT_CGY_TP_CODE'].mode()))

    # if mrcht_cite name does not have a category unkown, make
        one and the use to impute null values
    # if it does have one, use it to imute null values
    if 'UKN' in purchases['MRCHT_CTY_CODE'].cat.categories:
       df['MRCHT_CITY_NM'] = df['MRCHT_CITY_NM'].fillna("UKN")
    else:
       df['MRCHT_CITY_NM'] =
           df['MRCHT_CITY_NM'].cat.add_categories('UKN')
       df['MRCHT_CITY_NM'] = df['MRCHT_CITY_NM'].fillna("UKN")

    #keep only certain variables

    df = df[["TXN_ID",
           "CC_AR_ID",
           "CST_ID",
           "CRD_ID",
```

```
            "TXN_DATE_CON",
            "TXN_AMT",
            "MRCHT_NM",
            "MRCHT_CTY_CODE",
            "CAMS_MRCHT_CGY_TP_CODE",
            "MRCHT_CITY_NM"]]
    return df

def feature_engineering(customer,fraud,purchases):
    """Creates feature matrix

    Inputs
    ---------
    customer = cleaned and imouted file of all customer -
        target entity
    fraud = label table containing date of reported fraud and
        target variable
    purchases = cleaned and imputed purchase table

    Returns
    ---------
    Feature matrix and list of features produced

    """


    #creates enity set called
    es = ft.EntitySet(id = 'fraud')
    # adds customer dataframe to entity set and specifes index
        cst_id
    # specifies which varibales are categories
    es.entity_from_dataframe(entity_id='customers',
        dataframe=customer,
                    index = 'CST_ID',
                    variable_types = {'AGE': vtypes.Categorical,
                                 'HOGAN_CST_TP_DESC':
                                     vtypes.Categorical})

    # adds purchases dataframe to entity set and specifes index
        txn_id and time index TXN_date_con
    # specifies which varibales are categories
    es.entity_from_dataframe(entity_id='purchases',
        dataframe=purchases,
                    index = 'TXN_ID',
                    time_index = 'TXN_DATE_CON',
                    variable_types = {"CC_AR_ID":
                        vtypes.Categorical,
                                 "CRD_ID": vtypes.Categorical,
                       'MRCHT_NM':vtypes.Categorical,
                       'MRCHT_CTY_CODE':vtypes.Categorical,
                       'CAMS_MRCHT_CGY_TP_CODE':vtypes.Categorical,
                       'MRCHT_CITY_NM':vtypes.Categorical })

    #defines how customer and purcahse tables are related
```

```
    r_purchases = ft.Relationship(es['customers']['CST_ID'],
        es['purchases']['CST_ID'])
    #adds the realtionship to entity set
    es.add_relationships([r_purchases])

    #defines a cut off time from the fraud data
    cutoff_time = fraud[["CST_ID","CUT_OFF_TIME"]]

    #defines transformative primitives
    trans_primitives = ['is_weekend',"weekday", 'cum_sum',
        'day', 'month', 'diff', 'time_since_previous']
    #defines aggreagtion primitives
    agg_primitives = ['sum', 'time_since_last',
        'avg_time_between', 'mode', 'num_unique', 'min',
               'mean', 'max', 'std', 'count']
    #rund the dfs algorithm with specfied cutt-off times,
        primititeve and target entitity
    feature_matrix, feature_defs = ft.dfs(entityset=es,
        target_entity='customers',
                    cutoff_time =cutoff_time,
                    agg_primitives = agg_primitives,
                    trans_primitives = trans_primitives,
                    max_depth=2, features_only=False)
    return (feature_matrix, feature_defs)



def partition_to_feature_matrix(partition, write = True):
    """Take in a partition number, create a feature matrix, and
        save to Amazon S3


    Params
    --------
       partition (int): number of partition being used
       write: (boolean): whether to write the data to S3.
           Defaults to True

    Return
    --------
       None: saves the feature matrix to Amazon S3

    """
    #sets the parition directory
    partition_dir = BASE_DIR + 'p' + str(partition)

    # Read in the data files from parititon
    #read in purchases and spefcies dates and cateories
    purchases = pd.read_csv(f'{partition_dir}/purchases.csv',
                parse_dates=['txn_date_con', 'date_rcvd'],
                 infer_datetime_format = True,
                 dtype = {'cc_ar_id':'category',
                        "crd_id": 'category','MRCHT_NO':
                            'category',
```

```
                    'mrcht_nm':'category',
                    'mrcht_cty_code':'category',
                    'cams_mrcht_cgy_tp_code':'category',
                    'cams_mrcht_cy_tp_desc':'category',
                    'mrcht_str_adr':'category',
                    'mrcht_city_nm':'category',
                   })
    #read in customer and spefcies dates and categories
    customer = pd.read_csv(f'{partition_dir}/customer.csv',
           infer_datetime_format = True,
                  dtype = {'age':
                      'category','hogan_cst_tp_desc':'category'})
    #read in lable data
    fraud = pd.read_csv(f'{partition_dir}/fraud.csv',
        parse_dates=['cut_off_time'],
                infer_datetime_format = True)
    #double sur ethat the cst_id is a integer
    purchases['cst_id'] = purchases['cst_id'].astype('int64')

    #clean and impute the customer info
    customer = customer_preprocess(customer)
    #Clean and imute the purcashes data
    purcashes = purchases_clean(purchases)
    #applies the feature engieering function to create a
        feature matrix
    feature_matrix, feature_defs =
        feature_engineering(customer,fraud,purchases)

   #write feature martix back to S3 bucket

    if write:
       # Save to Amazon S3
       bytes_to_write = feature_matrix.to_csv(None).encode()

       with fs.open(f'{partition_dir}/feature_matrix.csv',
           'wb') as f:
          f.write(bytes_to_write)


def combine_across_s3(filename, name, last_partition):
    """ Combines a dataframe acorss the paritions and writes
        back to S3

       Inputs
       --------
       filename - file name to read in and combine
       naem - name for output file
       last_partition - last partition you want to combine

       Return
       --------
       Single combine csv file with info across the S3 paritions

    """
```

```
    #set base directory
    BASE_DIR = 's3://masterdatafinal/finalresults'
    #iterate across the paritions
    #adds another iteration for write out step
    for i in list(range((last_partition+1))):
        #sets the pariton file
        partition_dir = BASE_DIR + 'p' + str(i)
        #reads in csv file
        df = pd.read_csv(f'{partition_dir}/{filename}.csv')
        #if i is 0, the combine file is itself
        if i == 0:
            combine = df
        #for all other files, append data
        if i >0 and i < last_partition:
            combine = combine.append(df)
        #write out fil
        if i == last_partition:
            bytes_to_write = combine.to_csv(None).encode()
            with fs.open(f'{BASE_DIR}/{name}.csv', 'wb') as f:
                f.write(bytes_to_write)

#set total number of parititon
N_PARTITIONS = 1000
#set base directory where paritions sit
BASE_DIR = 's3://masterdatafinal/'

from timeit import default_timer as timer

start = timer()
partition_to_feature_matrix(836)
end = timer()
print(f'{round(end - start)} seconds elapsed.')

#create iterable from total number of paritions
partitions = list(range(N_PARTITIONS))

# use the sc.parallelize function to process the fucntin
    across the s3 partitions
# map to the nodes and collect the action command to force the
    trnasfomation to act
r = sc.parallelize(partitions, numSlices=N_PARTITIONS).\
   map(lambda x: partition_to_feature_matrix(x)).collect()

combine_across_s3('feature_matrix', 'features', N_PARTITIONS)

combine_across_s3('fraud', 'labels', N_PARTITIONS)
```